VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Perzistentní XML DOM

Persistent XML DOM

2013 Jaroslav Rozehnal

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Jaroslav Rozehnal**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Perzistentní XML DOM
Persistent XML DOM

Zásady pro vypracování:

Efektivní dotazování XML dat je aktuální výzkumný problém v oblasti databázových systémů. Z tohoto důvodu byla v poslední době vyvinuta celá řada datových struktur pro efektivní uložení XML dat. Jedním ze základních prvků nativní XML databáze je tzv. perzistentní DOM. Jedná se o perzistentní datovou strukturu modelující XML strom.

1. Nastudujte přístupy pro implementaci perzistentního DOMu.
2. Zvolený přístup implementujte.
3. Proveďte experimenty a vyhodnoťte je.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího diplomové práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **doc. Ing. Michal Krátký, Ph.D.**

Datum zadání: 18.11.2011
Datum odevzdání: 07.05.2013

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 30. 7. 2013 .....

Rád bych na tomto místě poděkoval svému vedoucímu doc. Ing. Michalu Krátkému, Ph.D. za odborné vedení a rady při tvorbě této práce. Dále bych také rád poděkoval svým rodičům a přítelkyni za podporu a pochopení v době, kdy jsem se jim kvůli svému vytížení nemohl věnovat.

Abstrakt

Tato práce prověřuje možnost využití persistentního pole jako klíčové komponenty při realizaci persistentního DOM. Je zde navržena a realizována vlastní implementace persistentního DOM, jež kromě persistentního pole využívá i B-stromu a číslovacích schémat Range order, Containment order a Dewey order pro identifikaci XML uzlů. Závěr práce představuje srovnávací testování, které hodnotí výkonnost persistentního DOM v závislosti na užití zmíněných číslovacích schémat a způsobů vyhodnocení dotazů. Toto řešení je také porovnáno s implementací DOM z knihovny Xerces.

Klíčová slova: PDOM, Persistentní pole, XML, B-strom, Xerces, Containment order, Dewey order, Range order

Abstract

This thesis evaluates usage of persistent array as a key component of persistent DOM implementation. Proposes my own implementation of persistent DOM using not only persistent array but also B-tree and ordering schemes Range order, Coontainment order and Dewey order as means to identify XML nodes. At the end there are presented comparsion tests, evaluating persistent DOM throughput based on usage of ordering schemes and ways of evaluating queries. This solution is also compared with DOM implementation from Xerces library.

Keywords: PDOM, Persistent array, XML, B-tree, Xerces, Containment order, Dewey order, Range order

Seznam použitých zkratek a symbolů

API	–	Rozhranní pro programování aplikací.
CDATA	–	Část XML souboru, která umožňuje vložení libovolných dat.
COM	–	Technologie programových komponent od firmy Microsoft.
DOM	–	Objektový model dokumentu, která reprezentuje rozhraní pro práci s XML nebo HTML dokumentem.
DTD	–	Jazyk pro popis struktury XML či SGML dokumentu.
EMC	–	Firma, která vyprodukovala jednu z XML databází.
GMD-IPSI XQL engine	–	Druh rozhraní pro práci s velkými XML dokumenty.
ISX	–	Engine pro uložení XML popsaný jedním z citovaných článků.
JAXP	–	Programové rozhraní jazyka Java pro práci s XML dokumenty.
LRU	–	Druh cachování informací, kdy je cachováno X naposled použitých záznamů.
Libmxml2	–	Knihovna pro práci s XML dokumenty.
MSXML	–	Knihovna pro práci s XML dokumenty.
PDOM	–	Persistentní objektový model dokumentu, která reprezentuje rozhraní pro práci s XML nebo HTML dokumentem. Použito také pro označení implementace základní funkčnosti PDOM realizované v této práci.
PHP	–	Serverový skriptovací jazyk.
RAM	–	Dočasná velmi rychlá vnitřní paměť počítače.
SAX	–	Jednoduché programové rozhraní pro práci s XML soubory. Je výrazně jednoduší než DOM.
SAX2	–	Přepracovaná novější verze SAX.
SDOM	–	Implementace persistentního DOM popsaná v jednom z citovaných článků.
SDOM-CT	–	Varianta SDOM, která podporuje komprimaci dat.
SGML	–	Standardizovaný značkovací jazyk na jehož základě vznikly jazyky HTML a XML.
SOM	–	Objektový model pro definici schéma XML souborů.
StAX	–	Programové rozhraní pro čtení a zápis XML dokumentů.
TPM	–	Algoritmus pro ověřování vnímaného pořadí uzlů ve stromech.

W3C DOM	–	Specifikace DOM organizace W3C.
XMARK	–	Projekt pro testování XML řešení.
XML	–	Rozšiřitelný značkovací jazyk.
XML DOM	–	DOM XML souboru.
XPATH	–	Jazyk řešící identifikaci a dotazování na uzly v XML souborech.
XSD	–	Jazyk pro zápis schéma XML souboru.
XSLT	–	Jazyk pro transformaci XML souborů.
XercesDOM	–	Implementace základní funkčnosti pro dotazování pomocí knihovny Xerces představené v rámci této práce.
xDB	–	XML databáze od firmy EMC.

Obsah

Seznam tabulek

Seznam obrázků

Seznam algoritmů

1 Úvod

V současné době nás v prostředí internetu a počítačů obklopuje XML ze všech stran. Používá se jako univerzální datový formát - je použit v konfiguračních souborech programů, pro uložení dokumentů, pro zápis skriptů komplexních aplikací, či dokonce pro komunikaci mezi počítači. XML neboli eXtensible Markup Language (rozšiřitelný značkovací jazyk) není však striktně vzato samostatným datovým formátem, přestože jej tak mnoho lidí chápe, ale spíše specifikací, kterou je možno pro definici konkrétního datového formátu použít. Zjednodušeně tedy můžeme říci, že XML je základem pro vytvoření konkrétního formátu. Právě tato variabilita a možnost s daty ve formátu XML efektivně pracovat jak programově, tak i manuálně, vedla k jeho širokému rozšíření.

Dalším fenoménem této doby jsou takzvaná *Big data*. Tímto pojmem je označována situace, kdy velikost a množství dat, které je třeba zpracovat, narůstá do obrovských rozměrů. Například anglická Wikipedia [24] uvádí maximální velikost souvisejících dat zpracovatelných v přijatelném čase až o řádu exabytů (což je 1024^6 bajtů). Tento fenomén zvyšuje nároky na zpracování dat a přináší nové druhy problémů, kterými se programátoři musí zabývat.

Fenomén *Big data* se samozřejmě projevuje i ve světě *XML* technologií, i když ne nutně stejně dramaticky jako v jiných oborech. Dochází nejen k nárůstu velikosti zpracovávaných dokumentů, ale často i k nárůstu jejich množství a potřebě hromadného zpracování, jelikož spolu jednotlivé dokumenty úzce souvisí. Z tohoto důvodu vznikají implementace XML databází a rozhraní pro práci s XML dokumenty, které nejsou limitovány omezenou a relativně drahou operační pamětí počítače. Další vývoj související s touto problematikou je pozorovatelný i na úrovni paralelního zpracování a distribuovaných výpočtů, ty však s touto prací již nesouvisí.

Diplomová práce se zabývá právě jednou z činností související s uvedenými fenomény. Pojem persistentní XML DOM označuje právě zmíněné rozhraní pro práci s XML dokumenty. Zatímco klasické DOM tradičně načítá celý zpracovávaný dokument do objektového modelu v paměti počítače, čímž značně omezuje jeho maximální velikost, persistetní DOM vytváří objektový model na pevném úložišti počítače, které je díky mnohem menší ceně za 1 GB mnohem dostupnější.

1.1 Cíl této práce

Tato práce se zaměřuje na využití speciální struktury, tzv. persistentního pole, jako klíčové komponenty při implementaci persistentního DOM. Persistentní pole je stránkovaná datová struktura, jejíž hlavní výhodou je efektivní sekvenční čtení. V rámci práce se využívá faktu, že právě sekvenční čtení je výhodné při implementaci většiny DOM funkcí při dotazování XML souboru.

Cílem této práce je ověřit vhodnost využití persisteního pole jako hlavního úložiště především v kontextu dotazování. Práce se nezabývá tvorbou XML souborů ani jejich pozdější aktualizací, řeší pouze jejich čtení. Proto je kladen důraz nejenom na vlastní vyhodnocení dotazu do persistentního DOM, ale také na efektivitu rozparsování vstup-

ního XML do datového úložiště. Možnost a efektivita úpravy již existujícího modelu dle aktuálních pokynů uživatele však není nijak řešena.

1.2 Popis struktury práce

Tato práce je uvedena kapitolou 2 XML technologie, ve které zmiňuji stručnou historii XML, shrnuji některé běžně používané technologie a popisuji několik existujících implementací DOM zaměřených na velké XML soubory. V kapitole 3 pojmenované jako Návrh persistentního DOM je navržen postup pro zpracování velkého XML a následně se v kapitole 4 Implementace persistnetního DOM věnuji detailům některých významných milníků implementace. V kapitole 5 Testování a srovnání je představeno výkonnostní testování persistentního DOM. Kapitola přináší srovnání této práce s referenčním řešením - implementací založené na klasickém DOM, ale vyhodnocuje i efektivitu jednotlivých nastavení samotného persistentního DOM. Práci uzavírá kapitola 6 Závěr shrnující dosažené cíle a hodnotící výsledky.

2 XML technologie

2.1 Historie XML

XML, eXtensible Markup Language [10], vznikl koncem devadesátých let dvacátého století jako produkt činnosti pracovní skupiny o jedenácti členech s podporou přibližně stoadesátičlenné zájmové skupiny. K vedoucím pracovní skupiny patřili Jon Bosak, Tim Bray, James Clark. XML bylo vytvořeno zjednodušením, zpřehledněním a doplněním poměrně komplikovaného jazyka SGML, kterým se však dalo již tehdy dosáhnout podobných výsledků, k nimž je dnes používáno XML.

2.2 XML DOM

DOM (Document object model) [11] je jazykově a platformově nezávislé rozhraní, které umožňuje dynamický přístup a úpravu v struktuře a stylu dokumentů založených na značkovacích jazycích. Hovoříme-li o XML DOM, jedná se o objektový model navržený speciálně pro práci nad XML dokumenty. Objektový model může být v počítači realizován několika způsoby, nejčastějším z nich je model založený na ukazatelích. V tomto modelu jsou všechny vazby mezi jednotlivými objekty - uzly reprezentovány pomocí ukazatelů. Chceme-li popsat nejběžnější způsob práce s XML dokumentem pomocí DOM, lze ji vystihnout následujícími kroky:

1. Rozparsování textové reprezentace XML a převod do objektového modelu.
2. Provedení prací nad objektovým modelem.
3. Přepsání původní textové reprezentace XML reprezentací novou vygenerovanou z objektové reprezentace.

Jak lze rozpoznat z předchozího seznamu kroků, je při typické práci s XML vždy vytvořena objektová reprezentace v paměti počítače, která se při ukončení práce uloží, což umožňuje efektivní práci s XML souborem běžných velikostí (v jednotkách až desítkách MB).

2.3 Existující implementace XML DOM

DOM bylo od svého vzniku mnohokrát implementováno v různých podobách pro rozličné jazyky. Zde se zmíníme o nejznámějších a také nejčastěji používaných implementacích DOM. Apache Xerces [20], [12] je populární implementace XML DOM spravovaná Apache Software Foundation. Jedná se o kolekci knihoven pro různé jazyky, mezi nimiž nechybí C++, Java ani Perl. Apache Xerces podporuje širokou řadu standardních API, mezi které patří i SAX2 či JAXP. Xerces je navržen pro vysokou výkonnost, spolehlivost, jednoduchost užití a snadnou portovatelnost mezi různými jazyky.

Libxml2 [21] je XML parser pro jazyk C vyvinutý v rámci projektu Gnome. Libxml2 představuje volně šiřitelný software licencovaný pod MIT licencí. Libxml2 je snadno portovatelný na řadu operačních systémů (Linux, Unix, Windows, MacOS, OS/2 a jiné),

implementuje řadu standardů, mezi které patří XML 1.0, XPath, XPointer, XInclude a obsahuje řadu rozšíření implementujících další standardy, např. XSLT.

Za zmínku dále stojí XML implementace vytvořená již v počátcích XML standardu firmou Microsoft. MSXML [22], [13] je implementováno v podobě COM objektů používaných napříč aplikacemi v MS Windows. Implementuje klasický W3C DOM standard, SAX, XSD, XSLT a SOM. V současné době je však spíše v pozadí především díky příchodu .NET frameworku, který obsahuje vlastní implementaci XML DOM.

2.4 Alternativy ke klasickému DOM

V současné době existuje několik alternativních přístupů pro práci s XML [18] [19]:

1. Proudově orientované parsery:
   (a) událostmi řízený parsing,
   (b) pull-parsing.
2. Mapování dat (data binding).

Událostmi řízený parsing [18] [19], jehož typickým příkladem je SAX (Simple Api for Xml) parser, zpracovává dokument sériově a obsah dokumentu oznamuje aplikaci pomocí zpětného volání funkcí (událostí). Tento parser bývá velmi jednoduše implementovatelný a efektivní při sekvenčním zpracováním dokumentu, přičemž neklade omezení na velikost souboru. Nicméně efektivita tohoto přístupu selhává, pokud se musí přistupovat k elementům v souboru náhodným způsobem. K alternativním příkladům řízeného parsování patří Expat parser.

Pull-parsing [18], [19] přistupuje k dokumentu jako k sérii položek, které jsou čteny sekvenčně za použití návrhového vzoru iterator. To umožňuje tvorbu rekurzivně sestupujících parserů, ve kterém struktura kódu zpracovávajícího data zrcadlí XML soubor. Příkladem pull parserů je StAX parser v Javě, XmlReader v PHP a System.Xml.XmlReader v .NET Frameworku.

Další formou API pro zpracování XML je mapování dat [18], [19], kde se data zpracovávají jako hierarchie uživatelských silně typovaných tříd v kontrastu k obecným třídám v DOM. Tento přístup zjednodušuje vývoj a v mnoha případech umožňuje identifikaci problémů již během kompilace. Příkladem systémů využívajících mapování dat je Java Architecture for XML Binding (JAXB) a XML Serialization v .NET Frameworku.

2.5 Persistentní DOM

Narozdíl od klasického DOM, které si bez problému vystačí s načtením celého dokumentu do počítače, je persistentní DOM nuceno využít komplexnější postup. I nepříliš velký soubor může totiž díky nutné redundanci dat v klasickém DOM snadno zahltit paměť počítače, a tak výrazně zpomalit zpracování dokumentu, nebo dokonce práci s dokumentem znemožnit. Tuto vlastnost persistentní DOM využívá zapojením pomalejšího, avšak násobně většího pevného disku, který umožňuje práci s XML soubory,

jejichž velikost je limitována pouze velikostí diskového úložiště. S použitím pomalejšího paměťového úložiště a s nárůstem objemu dat však souvisí několik problémů, které je třeba řešit.

Uvažujme naivní implementaci persistentního DOM, která by spočívala v uložení DOM XML dokumentu na pevný disk tak, aby odkazy použité v této struktuře směřovaly na umístění v souboru namísto umístění v paměti. Tímto postupem by bylo možné dosáhnout systému, který by byl ve své velikosti limitován pouze množstvím místa na pevném disku.

Nicméně již po krátkém zamyšlení napadnou čtenáře mnohé nevýhody takového řešení. Kromě nutnosti vytvořit interní reprezentaci před samotnou prací s XML, která je sice nutná i pro klasický DOM, ale díky rychlosti paměti RAM a velikosti souboru několikanásobně rychlejší, by práci velmi komplikovala nutnost přistupovat do souboru náhodným způsobem. Přestože je dnes náhodný přístup k pevnému disku poměrně efektivní, je stále podstatně pomalejší, než sekvenční čtení, kterému jsou pevné disky uzpůsobeny. Dalším znatelným omezením by byla nemožnost takovouto strukturu snadno aktualizovat. Každé přidání či odebrání elementu by totiž vyžadovalo přečíslování všech již existujících ukazatelů.

Pohled na možná řešení těchto a dalších problémů přináší následující kapitoly této práce.

2.6 Optimalizace DOM modelu

2.6.1 Rozdělení velkého XML souboru na menší části

Problémy nastíněné v kapitole 2.5 se zabývá [1]. Autoři v tomto článku navrhují DOM metodu pro získávání dat z velmi velkého XML dokumentu s dosažitelnou velikostí paměti a rozumnou dobou zpracování běžným osobním počítačem. Velký XML dokument je rozdělen do n malých dokumentů, kde n závisí na množství výpočetních zdrojů osobního počítače. Každý z těchto n malých dokumentů je následně modifikován zarovnávacím procesem (padding proccess) pro zajištění dobře formátovaného (well-formed) XML. Proces získání dat pomocí DOM API se následně vykonává sekvenčně na malých XML stromech sestavených ze všech modifikovaných n dokumentů. Výsledky ze všech n XML stromů jsou zkombinovány pro vygenerování konečného výsledku. S tímto přístupem mohou být operace získání dat z velmi velkých XML dokumentů vykonány na běžném osobním počítači.

[2] rozšiřuje [1] o procházení velmi velkých XML dokumentů. V [1] je velmi velký XML dokument D rozdělen do n malých XML dokumentů $< D1, D2, ...Dn >$, jsou získána data z každého DOM stromu zkonstruovaného z těchto n malých XML dokumentů a zkombinovány výsledky pro konstrukci jednoho sjednoceného DOM stromu. Požadovaná data jsou poté získána ze sjednoceného DOM stromu zkonstruovaného v hlavní paměti. Přistupovat takto k pomocným datům, která nejsou obsažena ve sjednoceném stromu, není jednoduché. Aby bylo možné získat jakákoli data jedním průchodem stromu, navrhuje se nový přístup založený na virtuálních uzlech a jejich asociativních operacích.

Autoři článku následně prezentují algoritmy pro vytvoření a správu virtuálních uzlů a představují čtyři základní operace pro navigaci v dokumentu - získání rodičovského uzlu (getParentNode), získání uzlů potomků (getChildNodes), získání předchozího sourozence (getPreviousSibling) a získání následujícího sourozence (getNextSibling). Tyto operace tak tvoří základ pro sestavení libovolných složitějších uzlů.

V [1] se představuje metoda pro práci s velkými XML soubory pomocí jejich rozdělení. Tato metoda je navrhována jako doplnění k standardním metodám zpracování, jelikož dosahuje vynikajících výsledků při práci s velkými XML dokumenty (kolem 1 GB), zatímco při práci s dokumenty menšími než 400 MB jsou dle testů autorů efektivnější standardní parsery. Vzhledem k povaze dělícího algoritmu je možné předpokládat zhoršení výkonnosti u velkých dokumentů, které nejsou tvořeny jako kolekce relativně malých XML sekvencí. Toto omezení však není třeba považovat za příliš velký problém - všechny současné opravdu velké XML soubory jsou tvořeny spojením množství menších. [1] implementuje pro přístup k dokumentům pouze metodu GET, která navrátí nalezený uzel na základě předané podmínky. Toto omezení je adresováno v [2], která dále rozšiřuje možnosti o navigaci v dokumentu při zachování stávající efektivity.

Ani jeden z článků se nezabývá možností úpravy dokumentu (tedy realizací metod INSERT, UPDATE a DELETE). Dále není dořešeno rozdělování dokumentu na části - oba články předpokládají externí definici počtu použitých částí.

2.6.2 Blízkostní DOM model

V [3] je navržena nová implementace DOM, označená jako SDOM, založená na datových strukturách pracujících na principu blízkosti (succintness), což přináší oproti běžným implementacím výhodu menší spotřeby paměti a v některých případech i rychlejší vykonání operací. To činí SDOM velmi vhodné pro reprezentování velkých statických dokumentů. SDOM implementuje téměř všechny funkce z DOM Level 3 Core API, vyjma operací modifikujících XML dokument.

Autoři dále popisují architekturu SDOM, jejímž jádrem je struktura nazvaná **S-tree**, která zachycuje stromovou hierarchii dokumentu. Popisují speciální techniky pro efektivní a úsporné uložení textových hodnot *CDATA* sekcí a dalších částí systému.

[3] představuje rychlou paměťovou reprezentaci dokumentu s malými paměťovými náklady. Ke zpracování velkých XML souborů je vhodné přistupovat podobně jako při jejich zpracování na velmi paměťově omezených počítačích (mobilní zařízení). Dále uvádí omezení algoritmu pouze na statické operace.

2.7 Optimalizace uložení dokumentu

Zatímco doposud jsme se věnovali optimalizaci DOM reprezentace dokumentu, případně načítání XML dokumentu z disku, nemalou pozornost je třeba věnovat i uložení vlastního DOM na disk.

2.7.1 Na blízkosti založené fyzické ukládací schéma

[4] představuje algoritmus vedoucí k zvýšení výkonu hledání elementů v XML dokumentu na základě jejich cesty.

Je založen na problému vyhodnocení výrazů cest (path expression) oproti XML stromům, jenž lze namodelovat obdobně jako problém hledání shody stromu vzorů (tree pattern matching - TPM). Výraz cesty (path expression) může být reprezentován jako strom vzorů, který specifikuje sadu omezení. TPM problém sestává z nalezení uzlů v XML dokumentu, které splňují všechna omezení.

Dále uvádí, že vyhodnocování a optimalizace výrazů cest spadá do dvou kategorií. *Navigační (Navigational)* přístup prochází strukturu stromu a testuje, zda-li uzel stromu splňuje omezení specifikovaná ve výrazu cesty. *Přístupy založené na operaci spojeni (join based)* nejdříve zvolí seznam uzlů XML stromu, které splňují omezení přiřazená k uzlu pro každý z uzlů stromu vzorů, a následně spojí párovaně seznamy na základě jejich strukturálních vztahů. S použitím řádných značkovacích technik (labeling techniques) lze vyhodnotit TPM s přijatelnou efektivitou různými technikami spojování (join techniques).

V článku je navržen nový přístup, který kombinuje výhody *navigačního* a na *na operaci spojení založeného (join based)* přístupu. Opodstatnění je založeno na pozorování, že některé strukturální vztahy implikují vyšší úroveň lokálnosti v XML dokumentu než jiné, a tak mohou být vyhodnoceny efektivněji využitím navigačního přístupu. Jiné na druhou stranu reprezentují globálnější vztahy, a tak mohou být vyhodnoceny efektivněji pomocí přístupu založeného na operaci spojení.

Na základě této ideje je definován strom vzorů nejbližšího příbuzného (next-of-kin - NoK), ve kterém jsou uzly propojeny pouze na základě vztahů rodič-dítě a předchozí/následující-sourozenec (což je označeno za lokální vztahy). Na základě obecného výrazu cesty je nejdříve rozdělen strom vzorů do propojených NoK stromů vzorů, na které jsou aplikovány efektivnější navigační algoritmy. Potom jsou výsledky hledání shody z NoK vzorů sjednoceny na základě jejich strukturálních vztahů, obdobně jak je tomu v přístupu založeném na spojení.

[4] představuje efektivní způsob uložení XML souboru, při jehož dotazování je omezen počet strukturálních spojení, což umožňuje efektivní vyhodnocování dotazů (vyžaduje pouze jeden průchod dokumentem). Autor dále uvádí možnost zlepšení efektivity v některé z dalších prací a potřebu implementace path-indexu namísto indexu dle jmen uzlů, který je v článku uveden.

2.7.2 Kompaktní ukládání XML

[5] se zabývá nalezením kompaktního schématu pro uložení XML, které bude prostorově efektivní reprezentací datové struktury se zachováním nízkých cen přístupu a aktualizace pro všechny požadované primitivní operace zpracování dat. Flexibilita XML činí nalezení schématu uspokojujícího všechny tyto požadavky zároveň velmi náročným.

Při hledání kompaktního schématu uložení XML existuje několik zásadních problémů:

1. Musí podporovat rychlé navigační operace.

2. Musí podporovat efektivní operace vkládání a mazání.

3. Musí podporovat efektivní operace spojení.

4. Musí být praktické.

5. Mělo by oddělit topologii, schéma a text dokumentu.

6. Mělo by umožnit dodatečné indexování.

V článku je dále navržen kompaktní engine pro uložení XML nazvaný ISX (podle Integrated Succint XML system) pro splnění všech výše zmíněných požadavků. ISX teoreticky používá množství místa blízké teoretickému minimu pro náhodné stromy. Pro konstantu ϵ, kde $1 <= \epsilon <= 2$, a dokument s n uzly, je potřeba $2 * \epsilon * n + O(n)$ bitů pro reprezentaci topologie XML dokumentu. Vkládání uzlů může být v průměru provedeno v konstantním čase a v nejhorším případě v čase O($lg^2 n$), operace navigace mezi uzly průměrně v konstantním čase a nejhůře v čase O($\frac{lgn}{lglgn}$).

V [5] je popsán kompletní systém pro uložení dat, jeho dotazování a modifikování. Autor dále uvádí výsledky testů srovnávající ISX s jinými systémy, včetně systému popsaném v kapitole 2.7.1, kde předvádí vyšší efektivitu svého systému.

2.7.3 Existující implementace persistentního DOM

V současné době existuje několik funkčních implementací persistentního DOM, přičemž se jedná zpravidla o komerčně vyvíjené knihovny.

V [17] popisuje autor implementační detaily eXist-db, open-source databáze postavené na jazyku Java. Jádro této databáze je tvořeno PDOM implementací vybudované nad upraveným číslovacím schématem, které nahlíží na strom XML dokumentu jako na úplný k-nární (například binární) strom. Toto číslovací schéma přiřazuje XML elementům rostoucí identifikátory dle faktoru *k*. [17] dálě popisuje modifikaci zmíněného číslovacího schématu použitého v eXist-db, která řeší omezení maximální velikosti stromu, jež plyne z jeho vlastností.

Další implementací persistent DOM je XML databáze xDB [14] firmy EMC, jež je vytvořena stejně jako předchozí knihovna pro jazyk Java. Knihovna, respektive XML databáze, poskytuje trvalé, na pevném disku založené úložiště pro velké XML dokumenty, k nimž poskytuje přístup pro čtení i pro zápis. xDB se liší od ostatních trvalých XML skladů (persistent XML stores) svou optimalizací na přímý DOM přístup. Navigační operace korespondující s běžnými DOM operacemi jsou vykonávány v konstantním čase srovnatelném s klasickými DOM implementacemi. Narozdíl od ostatních XML databází nevyužívá xDB číslovacích schémat, ale spoléhá se na použití indexů pro rychlé vyhledávání.

V rozmezí let 2000-2005 byly také vyvíjeny další implementace persistent DOM, jejichž vývoj však již skončil. Mezi ně patří open source implementace MonsterDOM, která byla součástí projektu Ozone-DB [16]. Tato implementace fungovala v podobě rozšíření DOM implementace OpenXML, přičemž pro uložení uzlů využívala relační databáze.

Dalším dnes již neaktivním projektem, který byl vyvíjen kolem roku 2000, je GMD-IPSI XQL engine [15]. Tento engine obsahoval PDOM implementaci, jež poskytovala funkce cachování, defragmentace, garbage collection, transakčního zpracování, komprese a vícevláknového přístupu. V té době se jednalo o poměrně kvalitní řešení. Byla vystavěna s využitím SAX parseru.

2.8 Číslovací schémata

Číslovací schémata umožňují efektivní značení XML uzlů, které reflektuje jejich vzájemné vztahy v zdrojovém dokumentu, jako jsou například vazby rodič-syn či sourozenec-sourozenec. Číslovací schémata tak v konečném důsledku vedou k úspoře paměťového místa i zvýšení efektivity algoritmů navigace mezi uzly.

Různá číslovací schémata pracují efektivně v různých situacích. Některá jsou efektivní při vyhodnocování navigace mezi XMl uzly, jiná mají extrémně malou paměťovou náročnost, zatímco další umožňují efektivní náhodné aktualizace DOM modelu.

Značné rozdíly jsou také mezi vlastnostmi uzlů ze zdrojového souboru, které číslovací schémata zachycují. Zatímco některá schémata pracují bezztrátově (tedy přesně reflektují všechny vztahy mezi uzly zdrojového dokumentu), jiná vybrané vlastnosti opomíjejí. Například číslovací schéma Range order neumožňuje rozlišit, je-li uzel přímým či nepřímým potomkem daného rodiče.

2.8.1 Range order (Rozsahové číslovací schéma)

Range order (rozsahové číslovací schéma) [6] využívá rozšířeného řazení preorder rozsahu potomků. Každému uzlu přiřazuje pár čísel `<order, size>` následovně:

1. Pro každý uzel y stromu a jeho rodiče x platí $order(x) < order(y)$ a zároveň $order(y)+size(y) <= order(x)+size(x)$. Jinými slovy interval $[order(y), order(y)+size(y)]$ je obsažen v intervalu $[order(x), order(x)+size(x)]$.

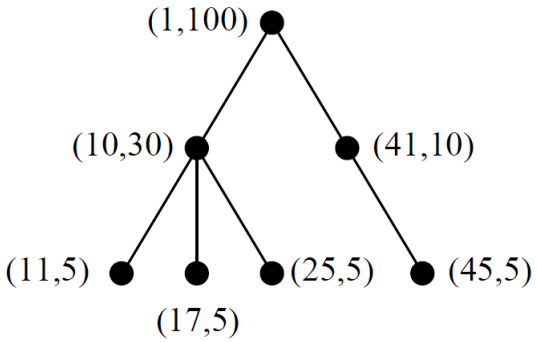
2. Pro každé dva sourozence x a y platí, že x je předchůdcem y pokud $order(x)+size(x) < order(y)$.

Potom pro každý uzel stromu x platí $size(x) >= \sum_y size(y)$ pro každé y, které je přímým potomkem x. Proto může mít $size(x)$ libovolně velkou celočíselnou hodnotu, což umožňuje alokovat místo pro budoucí vkládání. Schéma je znázorněno na obrázku 1 v podobě s rezervovaným místem pro budoucí vkládání nových uzlů.

2.8.2 Containment order (Obsahové číslovací schéma)

Containment order (obsahové číslovací schéma) [8] je podobné číslovacímu schématu Range order (viz 2.8.1). Narozdíl od Range order, jež tvoří pár `<order, size>`, je Containment order tvořeno trojicí `<startorder, endorder, level>`. Při bližším porovnání vlastností obou číslovacích schémat je zřejmé, že je obsahové schéma tvořeno rozšířením rozsahového a rozsahové je z obsahového zpětně odvoditelné.

Pro obsahové číslovací schéma platí mírně upravené vlastnosti rozsahového schématu:



Obrázek 1: Schématické znázornění číslovacího schématu Range order (převzato z [6])

1. Pro každý uzel y stromu a jeho rodiče x platí $startorder(x) < startorder(y)$ a zároveň $endorder(y) <= endorder(x)$. Jinými slovy, interval $[startorder(y), endorder(y)]$ je obsažen v intervalu $[startorder(x), endorder(x)]$.

2. Pro každé dva sourozence x a y platí, že x je předchůdcem y, pokud $endorder(x) < startorder(y)$.

3. Pro každý uzel stromu x platí, $size(x) >= \sum_y size(y)$ pro každé y, které je přímým potomkem x (pozn. $size(x)$ a $size(y)$ nejsou sice přímou součástí obsahového číslovacího schématu, je však možné je jednoduše odvodit). I obsahové číslovací schéma proto může jako $size(x)$ použít libovolně velkou celočíselnou hodnotu.

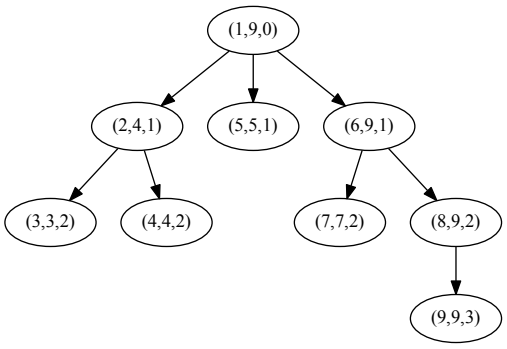
Třetí součást identifikátoru `level` označuje úroveň zanoření v XML stromu. Narozdíl od rozsahového číslovacího schématu tak lze nejenom určit, je-li uzel `x` rodičem uzlu `y` (pokud platí $startorder(x) < startorder(y)$ a zároveň $endorder(y) <= endorder(x)$), ale i zda-li je rodičem přímým (zároveň s předchozí podmínkou platí i $level(x)+1 = level(y)$). Tato vlastnost je velmi užitečná při dotazech na přímé potomky, jelikož je o tom možné rozhodnout již na základě porovnání číslovacích schémat bez nutnosti dalších, často časově náročných operací.

Znázornění tohoto schématu lze nalézt na obrázku 2.

Obdobně jako u Range order je i u Containment order možné ponechat v identifikátorech jednotlivých uzlů rezervu, která umožní snadné budoucí vkládání nových dat.

2.8.3 Dewey order (Dewey číslovací schéma)

Dewey order [7] je založeno na Dewey celočíselné klasifikaci (Dewey Decimal Classification) vyvinuté pro obecnou klasifikaci znalostí. V Dewey číslování je každému uzlu přiřazen vektor, který reprezentuje cestu od kořene dokumentu ke konkrétnímu uzlu. Každá komponenta cesty představuje lokální pořadí rodičovského uzlu tak, jak je znázorněno na obrázku 3. Dewey order je „bezztrátový“, jelikož každý identifikátor jedinečně určuje absolutní pozici uzlu v dokumentu.



Obrázek 2: Schématické znázornění číslovacího schématu Containment order

2.9 Persistentní pole

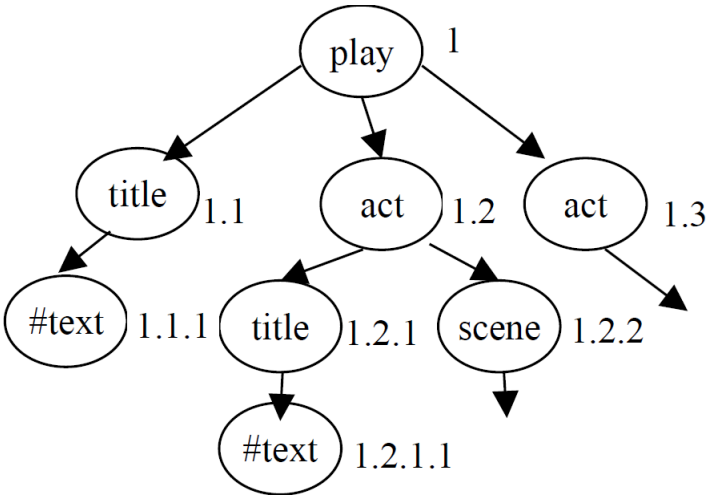
Persistentní proudové pole [9] je datová struktura využívající běžné stránkované schéma, ve kterém jsou uchovávány záznamy o uzlech XML dokumentu, a to v blocích ve vnějším úložišti. Pro zvýšení efektivity je použita cache ve vnitřní paměti, která uchovává bloky mimo vnější úložiště. Na obrázku 4 lze nalézt přehled popsaného schématu. Cache využívá přístupu naposledy použitých prvků (least recently used - LRU) pro výběr cachovaných bloků. Každý blok obsahuje pole dvojic - záznam o uzlu a ukazatel na další blok v proudu, což vytváří dynamický charakter této datové struktury. Jednoduše lze vložit či odebrat dvojice z bloku použitím rozdělení či sloučení uzlů. Bloky nemusí být plně využity, jelikož je současně uchováván i počet dvojic uložených v každém z bloku.

Vzhledem k proudové povaze persistentního pole je efektivní zejména sekvenční přístup k poli (jak pro čtení, tak i pro zápis). Přestože je efektivní i náhodný přístup, dochází v tomto případě obdobně jako u všech struktur uložených na pevném disku k degradaci efektivity.

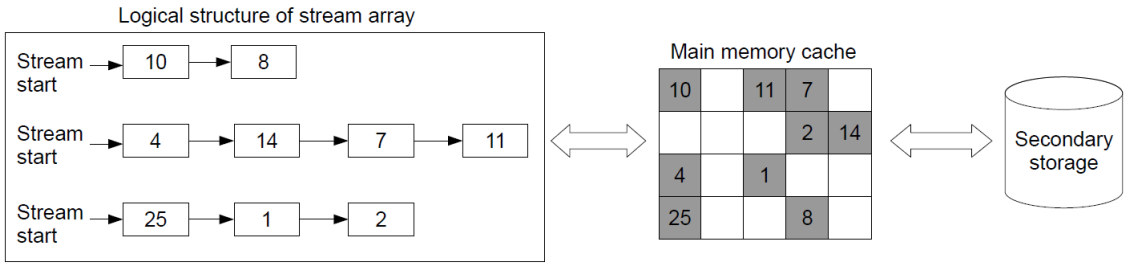
2.10 B-strom

B-stromy [23] představují vyvážené stromy, které jsou optimalizované pro situace, ve kterých je část či celý strom uchován ve vnějším úložišti, například magnetický disk. Vzhledem k tomu, že přístupy na disk jsou časově náročné operace, snaží se B-strom o snížení jejich počtu. Například B-strom o výšce dva s faktorem větvení 1001 může uložit přes milion klíčů, přičemž potřebuje maximálně dva diskové přístupy pro nalezení jakéhokoli uzlu.

Narozdíl od binárního stromu může mít každý uzel B-stromu proměnlivý počet klíčů a potomků. Klíče se ukládají v nesnižující se posloupnosti. Každý klíč je asociován potomkem, který je kořenem podstromu, jenž obsahuje všechny klíče s hodnotou menší než původní klíč, ale vyšší, než je následující klíč. Uzel má také dalšího potomka, který je uzlem pro podstrom obsahující všechny klíče větší než jakýkoli klíč uzlu. Vzhledem



Obrázek 3: Schématické znázornění číslovacího schématu Dewey order (převzato z [7])



Obrázek 4: Schématické znázornění stránkované datové struktury (převzato z [9])

k tomu, že má každý uzel tendenci mít velký faktor větvení, je většinou nutné procházení relativně malého množství uzlů pro nalezení vyhledávaného klíče.

3 Návrh persistentního DOM

Tato kapitola se věnuje návrhu algoritmů pro zajištění dvou ze základních operací:

1. **Naplnění datových struktur persistentního DOM** ze zpracovávaného XML souboru. V průběhu zpracování jsou data připravována pro pozdější dotazování.
2. **Dotazování datových struktur persistentního DOM** obsahující data ze zpracovaného XML souboru.

Návrh persistentního DOM je rozdělen do dvou kapitol, 3.1 a 3.2. Zatímco první kapitola se zabývá návrhem parsování vstupního souboru, druhá z kapitol navrhuje algoritmus pro vyhodnocování dotazů.

3.1 Parsování XML dokumentu

V průběhu parsování XML dokumentu dochází k rozdělení dokumentu na dílčí elementy, které označujeme jako uzly. Za uzly považujeme nejenom „skutečné“ uzly xml stromu, jenž jsou označeny počátečním a koncovým tagem (či tagem prázdným), ale také jejich hodnoty, atributy a hodnoty atributů. Komentáře a DTD specifikace jsou při zpracování vstupního souboru ignorovány.

Po identifikaci uzlů dochází k jejich označení jedinečným identifikátorem vytvořeným dle použitého číslovacího schématu, převedení do vhodné podoby a následnému uložení do datového úložiště.

3.1.1 Rozparsování XML dokumentů do uzlů

Dokument se nejdříve zpracovává parsovacím algoritmem. Vstupem parsovacího algoritmu bývá zpravidla název souboru či již načtený obsah, výstupem informace o uzlech XML stromu. Předání výstupních informací volajícímu kódu je realizováno různými způsoby, zpravidla voláním callback funkcí či jiných alternativ z parsovacího algoritmu.

Pro parsování je vhodné využít některou z již hotových a odladěných knihoven. Autor tak nejenom ušetří práci, ale také získá jistotu kvalitní implementace, kterou by jenom těžko dohánět vlastními silami.

Popsanou funkčnost realizuje aplikace pomocí speciálně navržené třídy *XmlBuilder*, jež slouží k abstrakci procesu zpracování vstupního souboru. Tato třída umožňuje využití libovolného parsovacího mechanismu, jelikož zapouzdřuje danou funkčnost. V této práci se využívá parser knihovny Xerces, pro jehož potřeby je implementována třída *XercesXmlBuilder*, dědící od třídy *XmlBuilder*. Nicméně mechanismus parsování je plně abstrahován, přičemž hypotetické použití jiného parseru by vyžadovalo vytvoření nové implementující třídy pro třídu *XmlBuilder*.

3.1.2 Číslování uzlů

Proces číslování uzlů spočívá v označení každého nově vznikajícího záznamu o uzlu vhodným jedinečným identifikátorem. Pro tvorbu identifikátorů se hodí zejména speci-

ální číslovací schémata, která jsou přizpůsobena struktuře XML souboru. Takto vytvořený identifikátor nejenže jednoznačně identifikuje daný uzel, ale reflektuje i vnitřní vazby XML souboru. Konečným důsledkem použití číslovacích schémat je možnost určit vztah mezi dvěma uzly pouze na základě znalosti jejich identifikátorů, bez nutnosti doplnění dodatečných vazeb na rodiče, potomky a sourozence.

3.2 Dotazování PDOM

Proces dotazování persistentního DOM slouží k efektivnímu získávání informací z předem připravené reprezentace XML souboru. Vlastní rozhraní pro dotazování může být definováno různými jazyky, jako jsou například XPATH či XQuery. Po podrobnější analýze výrazových prostředků těchto jazyků však dospěje čtenář k závěru, že pro poskytnutí všech způsobů dotazování na informace uložené v XML souboru postačí pouze omezené množství funkčnosti, jež v rámci této práce označujeme jako tzv. elementární operace. Ty jsou popsány v kapitole 3.2.1.

Dotazování je rozděleno mezi dvě funkční jednotky:

1. **reprezentace uzlu**,
2. **persistentní úložiště**.

Reprezentace uzlu poskytuje abstraktní rozhraní pro dotazování, jež je popsáno elementárními operacemi. Samotný algoritmus vyhodnocování elementárních operací využívá rozhraní **Persistentního úložiště**, které abstrahuje konkrétní datové struktury persistentního DOMu.

3.2.1 Elementární operace

Základní rozsah funkčnosti, nutný pro poskytnutí všech způsobů dotazování XML souboru, je specifikován nasledujícími operacemi:

1. **GetChildren** - slouží k získání všech přímých následníků uzlu, tzv. děti.
2. **GetChildrenByTagName** - slouží k získání všech dětí uzlu, které mají specifické jméno.
3. **GetAttributes** - slouží k získání všech atributů uzlu.
4. **GetSiblings** - slouží k získání všech sourozenců uzlu.
5. **GetDescendants** - slouží k získání všech potomků uzlu.
6. **GetDescendantsByTagName** - slouží k získání všech potomků uzlu se specifickým jménem.
7. **GetValueItem** - slouží k získání uzlu s hodnotou. Lze použít jak na klasické uzly, tak na atributy.

Metody **GetChildren**, **GetChildrenByTagName**, **GetAttributes**, **GetSiblings**, **GetDescendants**, **GetDescendantsByTagName** dále podporují filtrování podle hodnot.

Vzhledem k podstatě této práce, která spočívá v prozkoumání možnosti využití persistentního pole jako hlavního úložiště persistentního DOM a srovnání tohoto řešení s jinými alternativami, lze omezit definici rozhraní pro dotazování na uvedené elementární operace.

3.2.2 Reprezentace uzlu - třída XmlNodeRecord

Uzel v persistentní reprezentaci XML souboru je reprezentován třídou *XmlNodeRecord*, která nese všechny důležité informace o tomto uzlu. Obsahuje především jedinečný identifikátor uzlu vytvořený na základě číslovacího schématu. To umožňuje snadné určení vztahu uzlu s uzly jinými. Tato třída je podrobněji popsána v kapitole 4.1.1.

3.2.3 Datové úložiště - třída PersistentStorage

Datové úložiště, reprezentované třídou (viz *PersistentStorage*), poskytuje abstraktní přístup k datovým strukturám persistentního DOM. Jako hlavní úložiště nesoucí informace o XML uzlech je použito tzv. **persistentní pole** (viz 2.9).

Persistentní pole je doplněno dvěma instancemi **B-stromu** 2.10. První instance, pracovně označená jako *Index*, slouží k vyhledání pozice uzlu v persistentním poli na základě jeho identifikátoru. Druhá instance B-stromu je pracovně označena jako *Kořenový index*. Jsou v ní indexovány kořenové uzly zpracovaných souborů, což umožňuje rozparsování více vstupních souborů do jednoho fyzického úložiště.

3.2.4 Vyhodnocení elementárních funkcí

Algoritmus vyhodnocení elementárních funkcí je rozdělen dle míry abstrakce do dvou vrstev, které odpovídají třídám *XmlNodeRecord* a *PersistentStorage*. Třída *XmlNodeRecord* reprezentuje abstraktnější vrstvu, která není závislá na konkrétních strukturách použitých k uložení dat. Třída *PersistentStorage* tvoří konkrétnější vrstvu, jejímž cílem je abstrakce přístupu k datovému úložišti.

Vyhodnocení konkrétní elementární funkce je možné zjednodušeně popsat v následujících čtyřech krocích:

1. Vytvoření **zástupného identifikátoru** číslovacího schématu.
2. Vytvoření filtru na základě očekávaných vlastností vyhledávaného uzlu.
3. **Dotaz do B-stromu**, na který je vrácen odkaz na jeden či více záznamů.
4. Načtení každého ze záznamů z persistentního pole a ověření, že splňuje předem stanovené podmínky **filtru**.

Kroky 1 a 2 jsou realizovány v třídě *XmlNodeRecord*, zatímco kroky 3 a 4 ve třídě *PersistentStorage*.

Uvedený postup vyhodnocení umožňuje implementaci všech elementárních metod. Vyhledání a navrácení velkého množství elementů z B-stromu je však často zbytečně časově náročné. Většina elementárních metod totiž vyhledává potomky konkrétního uzlu a až následně provádí omezení dle jejich vlastností.

Vzhledem k tomu, že jsou data zpracovávána i ukládána v pre-order řazení, lze použít mnohem efektivnější způsob, který limituje počet operací při vyhledávání v B-stromu, a tak i počet přístupů k disku.

Druhý ze způsobů vyhledání využívá vlastností persistentního pole, které umožňuje efektivní sekvenční čtení záznamů. V B-stromu postačí vyhledat pouze rodičovský uzel a následné potomky číst sekvenčně z persistentního pole.

Sekvenční čtení potomků z persistentního pole lze ukončit jednoduchou podmínkou na vztah předek-potomek aktuálně zpracovávaného uzlu vůči uzlu referenčnímu.

Pro možnost porovnání jsou součástí algoritmu vyhodnocování dotazu oba zmiňované přístupy, jenž jsou pro jednoduchost označeny jako **vyhodnocení s preferencí B-stromu** a **vyhodnocení s preferencí persistentního pole**.

4 Implementace persistentního DOM

Implementace persistentního DOM se člení do několika kapitol. V kapitole 4.1 se popisují základní datové struktury použité v rámci práce, následně je v kapitole 4.2 rozpracováno číslování uzlů XML souboru pomocí různých číslovacích schémat. V kapitole 4.3 je poté popsán postup tvorby datové reprezentace persistentího DOM, aby kapitola 4.4 přinesla popis vyhodnocování dotazů do této datové struktury. Závěrem v kapitole 4.5 poskytuji informace o nevyřešených problémech a omezeních.

Implementace využívá framework QuickDB [25] implementovaného v Database Research Group, Katedry informatiky Vysoké školy báňské - Technické univerzity Ostrava.

4.1 Základní datové struktury

Tato kapitola doplňuje konkrétní implementační detaily struktur, které byly obecně nastíněny v kapitole 3.

4.1.1 XmlNodeRecord

Jak již bylo uvedeno v kapitole 3.2.2, reprezentuje třída *XmlNodeRecord* záznam o uzlu v persistentní reprezentaci XML souboru.

V diagramu na obrázku 5 jsou představeny vazby třídy *XmlNodeRecord* na její okolí. Třída *XmlNodeRecord* dědí od abstraktních třídy *XmlNode*, která definuje rozhraní pro práci s uzlem XML stromu. Tato třída (*XmlNode*) dále rozšiřuje abstraktní třídu *ElementaryFunction*, která definuje rozhraní pro práci s elementárními operacemi. Vzhledem k tomu, že nedílnou součástí třídy *XmlNodeRecord* je vazba na identifikátor tvořený dle číslovacího schématu, obsahuje instanci abstraktní třídy *Ordering*, jež jednoznačně označuje daný uzel. Pro přehlednost je práce s tímto identifikátorem oddělena do speciální třídy *OrderedEntry*. V uvedeném diagramu nejsou pro zvýšení přehlednosti uvedeny metody, které slouží k serializaci této třídy do persistentního pole.

Třída *XmlNodeRecord* obsahuje následující data:

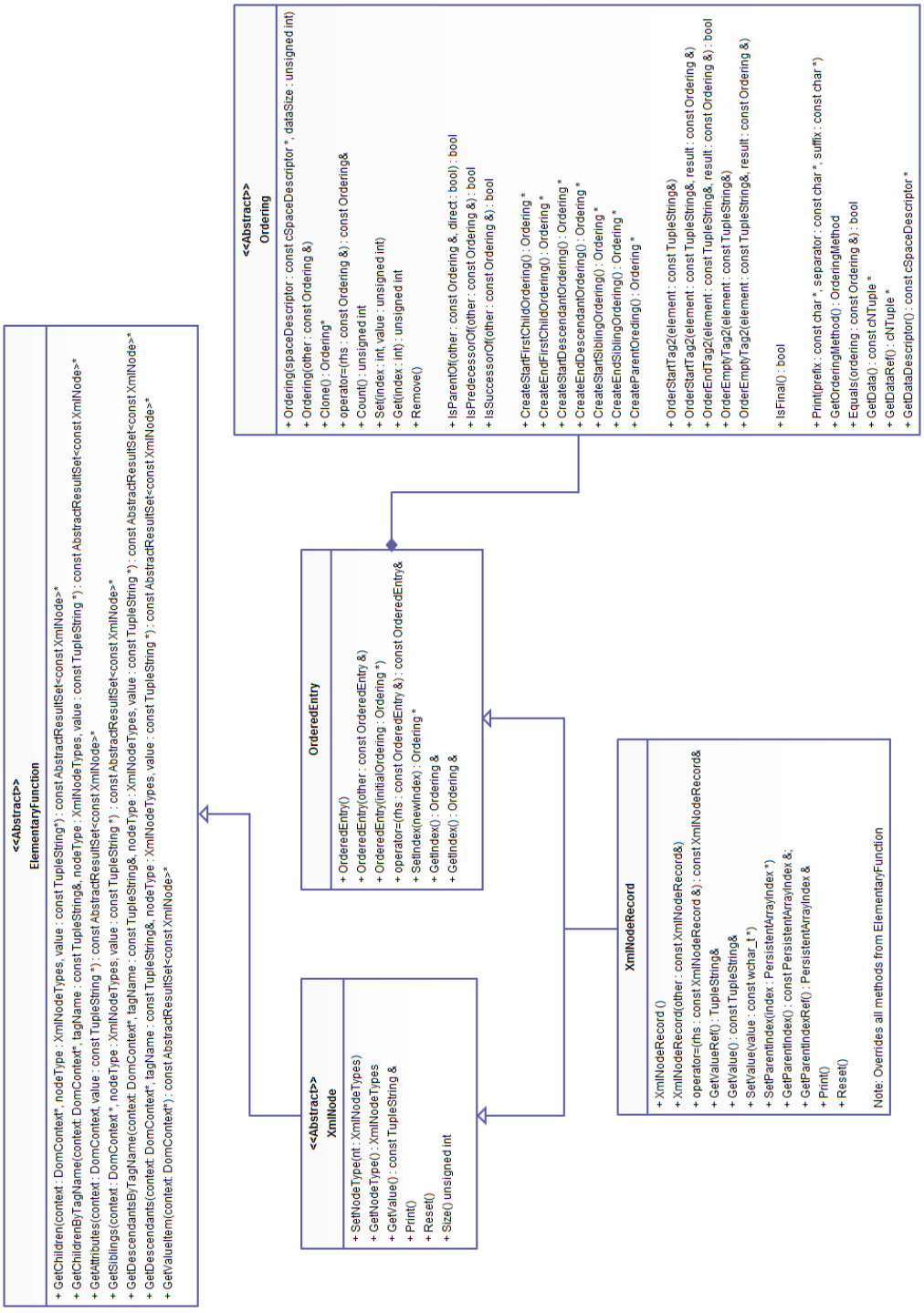
1. **identifikátor** vytvořený dle použitého číslovacího schématu, který označuje jedinečným způsobem uzel a umožňuje zároveň určit jeho vztah s ostatními uzly. Pro potřeby toho dokumentu je identifikátor libovolného implementovaného číslovacího schématu reprezentován třídou *Ordering*,

2. **typ**, jenž specifikuje jedná-li se o uzel, hodnotu či atribut,

3. **hodnotu**, která je interpretována jako název u záznamů typu uzel či atribut, a jako textová hodnota u záznamu typu hodnota,

4. **ukazatel na rodičovský uzel**. Ten slouží k doplnění chybějící informace u číslovacích schémat, které tuto informaci neuchovávají.

Tato třída dále implementuje elementárních funkce představené v kapitole 3.2.1.



Obrázek 5: Třídní diagram třídy XmlNodeRecord

4.1.2 Persistentní úložiště

Persistentní úložiště (třída *PersistentStorage*), navržené v kapitole 3.2.3, obsahuje tři struktury pro uložení informací o uzlech reprezentovaných třídou *XmlNodeRecord*:

1. **Persistentní pole** - úložiště, do kterého jsou ukládány jednotlivé záznamy o uzlech.
2. **Index** - B-strom, pro vyhledání záznamu o uzlu na základě jeho číslování.
3. **Kořenový index** - B-strom, pro vyhledání kořenového uzlu dle názvu vstupního souboru.

Třída *PersistentStorage* implementuje následující metody pro zpracování XML souborů a pro vyhledávání informací:

1. **Create** - vytvoří nové úložiště.
2. **Open** - otevře nový datový soubor persistentního úložiště.
3. **Close** - uzavře aktuálně otevřený soubor persistentního úložiště.
4. **Parse** - provede zpracování vstupního XML souboru do otevřeného persistentního úložiště.
5. **GetRoot** - vrátí kořenový uzel XML stromu pro daný rozparsovaný soubor.
6. **GetRecord** - načte uzel specifikovaný ukazatelem do persistentního pole.
7. **ReadFrom** - inicializuje čtení z persistentního pole na základě ukazatele do persistentního pole.
8. **SearchInBtree** - inicializuje čtení z persistentního pole vyhledáním v B-stromu.
9. **GetResults** - načte z persistentního pole uzly, které byly nalezeny předchozím dotazem do B-stromu.
10. **ReadUntil** - čte od pozice specifikované metodou **ReadFrom** či prvním naposledy nalezeným uzlem v B-stromu. Čtení je ukončeno nesplněním podmínky filtrování.

Vyhledání v B-stromu je realizováno na základě zástupných identifikátorů vytvořených dle použitého číslovacího schématu 4.4.1. Metody **GetResults** a **ReadUntil** využívají filtru 4.4.3 pro zpřesnění navrácených výsledků a v případě metody **ReadUntil** i specifikaci ukončovací podmínky.

Tato třída obsahuje některé další metody, které zde nejsou zmíněny. Jedná se však pouze o metody, jejichž účelem je podpora pro měření výkonnosti či pomoc pro ladění, a nejsou tedy autorem považovány za zajímavé při prezentaci zvoleného řešení.

4.2 Číslování uzlů pomocí číslovacích schémat

Pro podporu číslování uzlů jsou v rámci projektu implementovány tři číslovací schémata: *Range order* (rozsahové číslovací schéma), *Containment order* (obsahové číslovací schéma) a *Dewey order* (Dewey číslovací schéma).

Vztahy tříd implementující podporu pro číslovací schémata jsou uvedeny v třídním diagramu na obrázku 6. Instance identifikátoru číslovacího schématu je reprezentována abstraktní třídou *Ordering*. Tato třída je implementována ve třech dědících třídách *ContainmentOrdering*, *DeweyOrdering* a *RangeOrdering*, které implementují funkčnost pro podporu patřičného číslovacího schématu. Vytváření instancí je svěřeno abstraktní třídě *OrderingFactory*, jež implementuje návrhový vzor *Abstract factory*. Tato třída poskytuje rozhraní, pro implementující factory třídy, které se starají o vlastní vytváření instancí třídy ordering. Třída *OrderingFactory* poskytuje jednak metodu **CreateOrdering**, jež slouží k vytvoření nové instance třídy Ordering, a několik dalších metod, které umožňují zjistit vlastnosti a omezení aktuálně používaného číslovacího schématu.

Rozhraní specifikované třídou *Ordering* poskytuje několik skupin metod. Metody **Count**, **Set**, **Get** a **Remove** slouží k práci s jednotlivými částmi jedinečného identifikátoru, zatímco metody **IsParentOf**, **IsPredecessorOf** a **IsSuccessorOf** umožňují zjištění vzájemného vztahu dvou identifikátorů. Skupina metod **CreateStartFirstChildOrdering**, **CreateEndFirstChildOrdering**, **CreateStartDescendantOrdering**, **CreateEndDescendantOrdering**, **CreateStartSiblingOrdering**, **CreateEndSiblingOrdering** a **CreateParentOrdering** slouží k vytvoření tzv. zástupných identifikátorů (více viz kapitola 4.4.1).

Poslední skupinou jsou metody pro podporu tvorby identifikátoru při procesu parsování vstupního dokumentu. Metody **OrderStartTag2**, **OrderEndTag2**, **OrderEmptyTag2** slouží k realizaci činnosti při obsluze událostí parseru **StartNodeEvent**, **ValueEvent**, **EndNodeEvent** následovně:

1. Událost **StartNodeEvent** využívá metody **OrderStartTag2**.
2. Vzhledem k tomu, že hodnoty XML uzlů a atributů jsou interně reprezentovány samostatnými uzly, je při obsluze události **ValueEvent** využívána metoda **OrderEmptyTag2**.
3. Událost **EndNodeEvent** využívá metody **OrderEndTag2**.

Při procesu tvorby identifikátoru je dále využita metoda **IsFinal**, která umožňuje určení, je-li proces tvorby identifikátoru již dokončen a je-li identifikátor již použitelný při dotazování.

Vzhledem k tomu, že tato práce opomíjí možnost aktualizace persistentního DOM, jsou identifikátory uzlů tvořené podle číslovacích schémat *Range order* a *Containment order* ponechány ve své nejúspornější variantě, tedy bez rezervních míst pro budoucí uzly.

4.2.1 Range order (Rozsahové číslovací schéma)

Range order (Rozsahové číslovací schéma), jehož specifikace je popsána v kapitole 2.8.1, je velmi paměťově úsporným číslovacím schématem. Má několik nevýhod, této práce se dotýkají dvě z nich.

Vzhledem k povaze zachycení informace o vztazích mezi uzly není u *Range order* možné zjistit, je-li rodičovský vztah mezi uzly přímý či nepřímý. Pro získání této informace pouze na základě znalosti identifikátoru uzlu je nutné vyhledat všechny rodičovské uzly a provést jejich vzájemné porovnání. Druhým omezením, které se v rámci této práce řeší, je možnost vytvoření zástupného identifikátoru rodičovského elementu pouze na základě znalosti identifikátoru potomka. Řešení obou popsaných omezení navrhuji v rámci kapitoly 4.4.4.

Obsluha metod **OrderStartTag2**, **OrderEndTag2**, **OrderEmptyTag2** rozsahového číslovacího schématu je popsána v algoritmu 1, 2 a 3. Všechny metody sdílí společný zásobník a proměnné TagIndex a LastTag. V implementaci není zásobník součástí samotného algoritmu pro tvorbu identifikátoru, ale je sdílený pro všechny načtené uzly bez rozlišení použitého číslovacího schématu. Pro principiální popis algoritmu je však prezentován v rámci algoritmu pro tvorbu identifikátorů.

Algoritmus 1 Pseudokód popisující metodu OrderStartTag2 rozsahového číslovacího schématu

```
Zvýšení hodnoty čítače TagIndex o 1
Vložení na zásobník neúplný identifikátor <TagIndex, 0>
Uložení názvu aktuálně zpracovávaného uzlu do LastTag
```

Algoritmus 2 Pseudokód popisující metodu OrderEndTag2 rozsahového číslovacího schématu

```
Vyjmutí identifikátoru ze zásobníku
Pokud je název v LastTag stejný jako název aktuálního tagu:
Zvyš TagIndex o konstantu určující výchozí délku prázdného tagu
jinak:
Zvyš TagIndex o 1
Nastav identifikátoru velikost na TagIndex
vrať identifikátor
```

Algoritmus 3 Pseudokód popisující metodu OrderEmptyTag2 rozsahového číslovacího schématu

```
Zvyš TagIndex o jedna a ulož hodnotu do proměnné Start
Zvyš TagIndex o konstantu určující výchozí délku prázdného tagu
a ulož hodnotu do proměnné Size
Uložení názvu aktuálně zpracovávaného uzlu do LastTag
Vrať identifikátor <Start, Size>
```

4.2.2 Containment order (Obsahové číslovací schéma)

Číslovací schéma *Containment order* je velmi podobné rozsahovému číslovacímu schématu (viz kapitola 2.8.2). Obsahuje však navíc informaci o hloubce zanoření v XML stromu, která umožňuje rozlišit mezi přímým a nepřímým rodičem (potomkem). Stejně jako *Range order* však *Containment order* neumožňuje vytvořit zástupný identifikátor rodičovského uzlu, což je řešeno stejným principem jako u *Range order* (viz 4.4.4).

Obsluha metod **OrderStartTag2**, **OrderEndTag2**, **OrderEmptyTag2** rozsahového číslovacího schématu je popsána v algoritmu 4, 5 a 6. Všechny metody sdílí společný zásobník a proměnné TagIndex a LastTag. V implementaci není zásobník součástí samotného algoritmu pro tvorbu identifikátoru, ale je sdílený pro všechny načtené uzly bez rozlišení použitého číslovacího schématu. Pro principiální popis algoritmu je však prezentován v rámci algoritmu pro tvorbu identifikátorů.

Algoritmus 4 Pseudokód popisující metodu OrderStartTag2 rozsahového číslovacího schématu

```
Zvýšení hodnoty čítače TagIndex o 1
Zvyš hodnotu Level rodičovského uzlu o 1 a ulož ji do proměnné Level
Vložení na zásobník identifikátor <TagIndex, 0, Level>
Uložení názvu aktuálně zpracovávaného uzlu do LastTag
```

Algoritmus 5 Pseudokód popisující metodu OrderEndTag2 rozsahového číslovacího schématu

```
Vyjmutí identifikátoru ze zásobníku
Pokud je název v LastTag stejný jako název aktuálního tagu:
Zvyš TagIndex o konstantu určující výchozí délku prázdného tagu
jinak:
Zvyš TagIndex o 1
Nastav identifikátoru velikost na TagIndex
vrať identifikátor
```

Algoritmus 6 Pseudokód popisující metodu OrderEmptyTag2 obsahového číslovacího schématu

```
Zvyš TagIndex o jedna a ulož hodnotu do proměnné Start
Zvyš TagIndex o konstantu určující výchozí délku prázdného tagu
a ulož hodnotu do proměnné End
Zvyš hodnotu Level rodičovského uzlu o 1 a ulož ji do proměnné Level
Uložení názvu aktuálně zpracovávaného uzlu do LastTag
Vrať identifikátor <Start, End, Level>
```

4.2.3 Dewey order (Dewey číslovací schéma)

Číslovací schéma *Dewey order*, jehož specifikace je popsána v kapitole 2.8.3, je paměťově poměrně náročné číslovací schéma. Velikost identifikátoru je přímo závislá na hloubce zanoření označeného uzlu. Maximální velikost identifikátoru, které zpravidla musí být přizpůsobeno úložiště uzlů, je tak závislá na maximální hloubce zpracovávaného XML souboru, což může někdy způsobit omezení při použití tohoto číslovacího schématu. Toto číslovací schéma však umožňuje bezztrátové uložení informací o struktuře XML souboru. Lze tak nejen snadno určit jakoukoli vazbu, jež lze pozorovat u stromu XML souboru, ale i vytvořit zástupný identifikátor rodičovského uzlu.

Obsluha metod **OrderStartTag2**, **OrderEndTag2**, **OrderEmptyTag2** číslovacího schématu *Dewey order* je popsána v algoritmu 7, 8 a 9. Všechny metody sdílí jeden zásobník a proměnnou LastNumber, v které je uložena hodnota pořadí aktuálně zpracovávaného přímého potomka. V implementaci není zásobník součástí samotného algoritmu pro tvorbu identifikátoru, ale je sdílený pro všechny načtené uzly bez rozlišení použitého číslovacího schématu. Pro principiální popis algoritmu je však prezentován v rámci algoritmu pro tvorbu identifikátorů.

Algoritmus 7 Pseudokód popisující metodu OrderStartTag2 Dewey číslovacího schématu

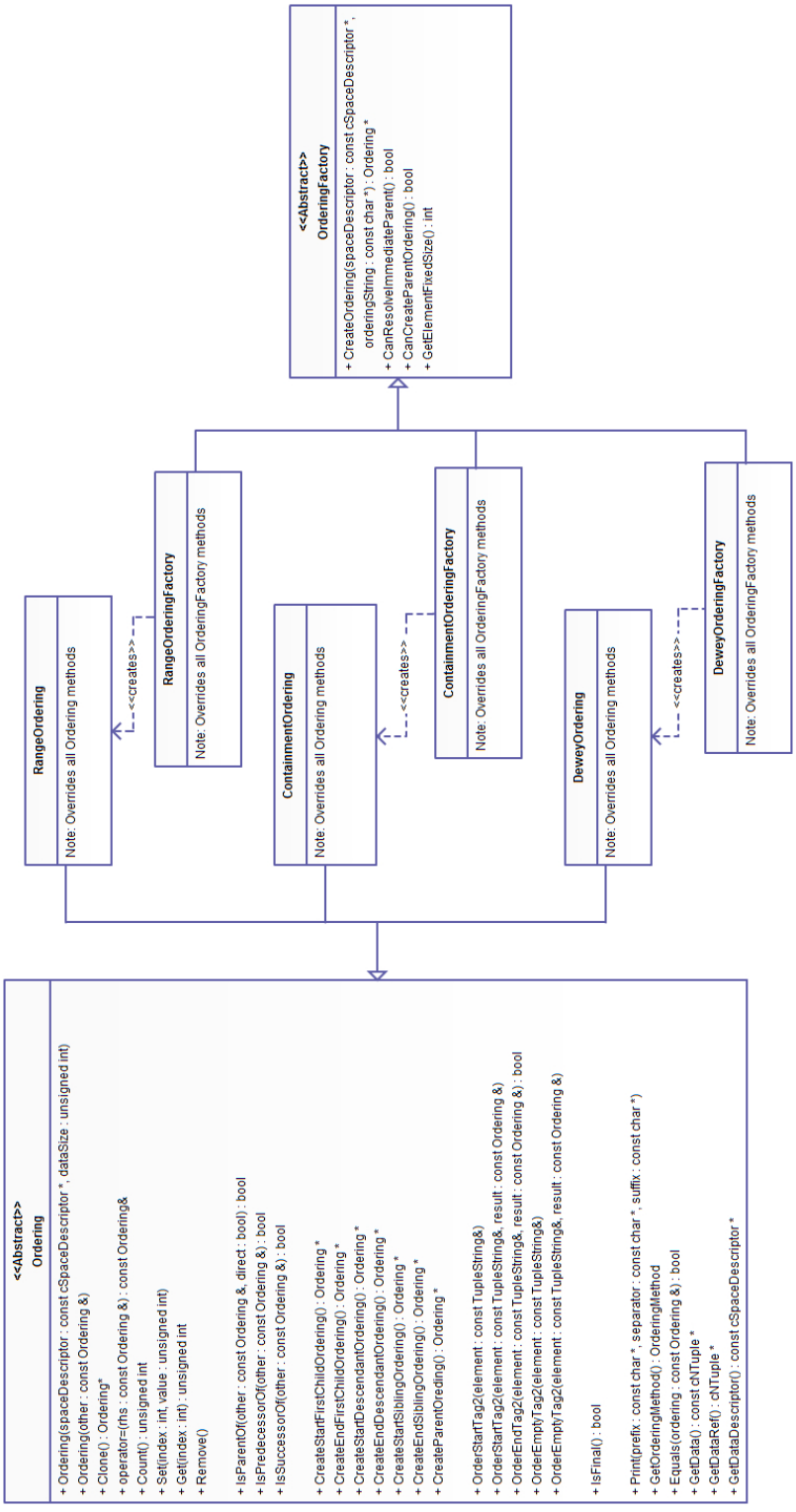
```
Vložení hodnoty LastNumber + 1 na zásobník
Vynulování hodnoty LastNumber
Vytvoření nového identifikátoru z hodnot na zásobníku
```

Algoritmus 8 Pseudokód popisující metodu OrderEmptyTag2 Dewey číslovacího schématu

```
Zvýšit hodnotu LastNumber o 1
Vytvoření nového identifikátoru z hodnot na zásobníku a z hodnoty
  LastNumber
```

Algoritmus 9 Pseudokód popisující metodu OrderEndTag2 Dewey číslovacího schématu

```
Vyjmutí hodnoty ze zásobníku a uložení v LastNumber
```



Obrázek 6: Třídní diagram pro třídy Ordering

4.3 Zpracování XML dokumentu

Tato kapitola představuje implementační detaily procesu parsování vstupního XML souboru do datových struktur persistentního úložiště.

Přestože je implementována podpora pouze jednoho parseru, umožňuje použitý návrh jednoduchou implementaci podpory pro jakýkoli jiný parser za předpokladu, že zpracovává uzly v preorder pořadí.

V rámci této práce je použit pro zpracování vstupního dokumentu Xerces parser.

Jak již bylo zmíněno v kapitole 3.1.1, reprezentuje abstraktní třída *XmlBuilder* rozhraní proxy tříd parsovacích algoritmů a jejich obslužného kódu. Diagram této třídy a tříd s ní souvisejících lze nalézt na obrázku 7. Rozhraní reprezentované třídou *XmlBuilder* se skládá ze tří klíčových metod **GetXmlMaxDepth**, **Build** a **GetRootNodeIndex**. Metoda **GetXmlMaxDepth** provede rychlý průchod parsovaným XML a zjistí jeho maximální hloubku, jejíž znalost je klíčová pro korektní inicializaci datových struktur při použití číslovacích schémat s proměnnou délkou identifikátoru, jako je například *Dewey order*. Metoda **Build** provede rozparsování dokumentu do datových struktur. Konstruktoru třídy *XmlBuilder* je při inicializaci předán ukazatel na třídu *BuilderContext*, která reprezentuje datové úložiště, do nějž jsou uzly zpracovávány. Alternativně lze pomocí metody **ResetContext** ukazatel na datové úložiště změnit.

Dalším částem diagramu na obrázku 7 se věnují kapitoly 4.3.1 a 4.3.2.

4.3.1 Parsování vstupního XML souboru závislé na parseru Xerces

Parsovací mechanismus využívající parseru z knihovny Xerces je implementován ve třídě *XercesXmlBuilder*. Třída dědí od třídy *XmlBuilder* a využívá wrapperu *cXmlXercesParser* pro realizaci vlastního parsování. Tento wrapper využívá abstraktní třídy *cStorageCreator_Abstract*, v které je specifikováno rozhraní pro jednotlivé události průchodu souborem. Abstraktní třídu *cStorageCreator_Abstract* implementuje třída *MaxDepthStorageCreator*, jež zajišťuje první průběh zjišťující maximální hloubku XML stromu, a třída *XercesStorageCreator*, jež realizuje plnění datových struktur.

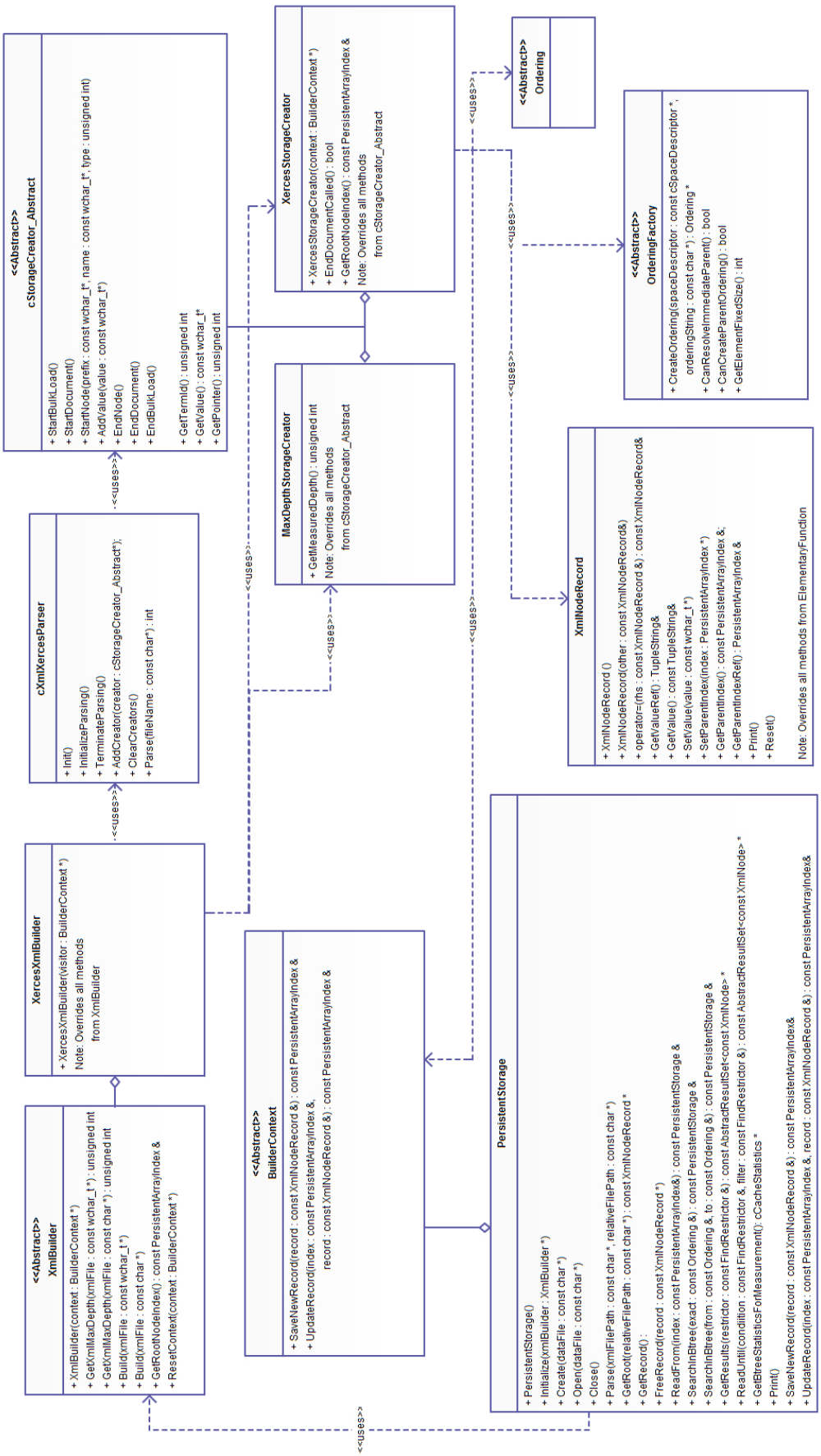
Třída *XercesStorageCreator* využívá tří klíčových metod zpracování XML souboru **StartNode**, **AddValue** a **EndNode**, jež odpovídají událostem zmíněným v kapitole 4.2. Tyto tři metody plně postačují pro zpracování vstupního XML souboru. Při obsluze těchto metod je využíváno tříd *OrderingFactory*, *Ordering*, *XmlNodeRecord* a *BuilderContext*.

4.3.2 Uložení záznamu do persistentního pole

Rozhraní úložiště záznamů o rozparsovaných XML uzlech je reprezentováno abstraktní třídou *BuilderContext*, která definuje dvě hlavní obslužné metody ukládájící vytvořené záznamy do datového úložiště: *SaveNewRecord* a *UpdateRecord*.

Metoda *SaveNewRecord* zajišťuje uložení nového záznamu do persistentního úložiště, zatímco metoda *UpdateRecord* může být volána pro aktualizaci již jednou uloženého záznamu, pokud to vyžaduje algoritmus tvorby identifikátoru číslovacího schématu.

Jednotlivé očíslované záznamy jsou ukládány do persistentního pole. Vlastní proces uložení využívá operace pro hromadné uložení, tak jak je popsáno v 3.2.3.



Obrázek 7: Třídní diagram třídy XmlBuilder

4.4 Dotazování datových struktur

Tato kapitola rozpracovává vyhodnocení elementárních operací, jež bylo představeno v části návrhu (v kapitole 3.2.1).

Dotaz na některou z elementárních operací je realizován na úrovni třídy *XmlNodeRecord*, která implementuje rozhraní specifikované abstraktní třídou *ElementaryFunction*. Třída *XmlNodeRecord* však nerealizuje vlastní vyhledání přímo, nýbrž využívá třídy *QueryEvaluator*, která abstrahuje vlastní proces vyhledání dle preferovaného způsobu vyhodnocení dotazu. Tato třída stejně jako třída *XmlNodeRecord* dědí od třídy *ElementaryFunction*. Vlastní vyhodnocení dotazu je implementováno ve třídách *BtreeQueryEvaluator* (implementuje vyhodnocení dotazu s preferencí využití B-stromu) a *ParrayQueryEvaluator* (implementuje vyhodnocení dotazu s preferencí využití persistentního pole) dědících od třídy *QueryEvaluator*.

Kapitola 4.4.1 se věnuje tvorbě zástupných identifikátorů, vyhodnocení vlastního dotazu je popsáno v kapitole 4.4.2 a kapitola 4.4.3 představuje omezení výsledku filtrováním.

4.4.1 Tvorba zástupných identifikátorů

Vyhodnocení dotazu s preferencí využití B-stromu využívá zástupných identifikátorů, které umožní vyhledání všech požadovaných uzlů jediným dotazem do B-stromu. Při dotazování preferujícím využití persistentního pole není třeba vytvářet zástupné identifikátory, jelikož je v B-stromu vyhledán pouze aktuální identifikátor a od něj je následně započato sekvenční čtení z persistentního pole.

Vyhodnocení dotazu preferujícího využití B-stromu je založeno na vlastnostech použitých číslovacích schémat. Při tvorbě zástupných identifikátorů je předpokládáno uplatnění následujících vlastností:

1. Lze vytvořit dvojici identifikátorů takovou, že identifikátor všech potomků vzorového uzlu je řazen mezi uvedenou dvojici.

2. Lze vytvořit dvojici identifikátorů takovou, že identifikátor všech sourozenců vzorového uzlu a jejich potomků je řazen mezi uvedenou dvojicí.

3. Lze vytvořit dvojici identifikátorů takovou, že v uvedeném rozsahu je zahrnut právě jeden uzel, jenž je zároveň první potomek vzorového uzlu.

Uvedené vlastnosti jsou klíčové pro vytvoření zástupných identifikátorů a korektního vyhodnocení dotazu do B-stromu. Některá číslovací schémata však druhý bod neumožňují, což je nutné obejít alternativním způsobem (více viz 4.4.4). V závislosti na použitém číslovacím schématu může být třetí vlastnost reprezentována pouze jedním identifikátorem, jenž označuje prvního potomka uzlu.

Třída *Ordering* definuje šestici metod pro získání zástupných identifikátorů jejichž implementace je přímo závislá na konkrétní vlastnosti.

1. Metody **CreateStartDescendantOrdering()** a **CreateEndDescendantOrdering()** spoléhají na vlastnost číslo 1.

Název metody	Range order	Containment order	Dewey order
GetStartDescendant(a)	2,X	2,X	1.1.1
GetEndDescendant(a)	4,X	4,X	1.1.MAX
GetFirstChild(e)	7,X	7,X	1.3.1
GetStartSibling(d)	-	-	1.1
GetEndSibling(d)	-	-	1.MAX

Tabulka 1: Příklady vyhodnocení typů číslování

2. Metody **CreateStartSiblingOrdering()** a **CreateEndSiblingOrdering()** spoléhají na vlastnost číslo 2.

3. Metody **CreateStartFirstChildOrdering()** a **CreateEndFirstChildOrdering()** spoléhají na vlastnost číslo 3.

Pro názornost je možné předvést funkci generování zástupných identifikátorů na příkladě. Graf na obrázku 9 zobrazuje jednoduchý ukázkový XML strom, ve kterém jsou uvedeny identifikátory dle všech tří použitých číslovacích schémat: číslovací schéma Range order je značeno *R()*, číslovací schéma Containment order je značeno *C()* a číslovací schéma Dewey order je značeno *D()*.

Tabulka 1 uvádí vybrané příklady jednotlivých zástupných identifikátorů dle použitého číslovacího schématu. V této tabulce jsou využity zástupné symboly, které jsou v praxi nahrazeny konkrétním číslem závisejícím na implementaci. Symbol *X* zastupuje libovolné číslo, na kterém díky vlastnostem použitého porovnávání nezáleží. Zástupný symbol *MAX* zastupuje maximální možné číslo reprezentovatelné v dané implementaci (obecně by se jednalo o hodnotu nekonečno). Znak „-“ označuje případy, ve kterých není možné pro použité číslovací schéma metodu zavolat a je třeba použít alternativní způsob vyhodnocení.

4.4.2 Vyhodnocení dotazu

Proces vyhodnocení výsledku dotazu je evaluován dvěma odlišnými způsoby v závislosti na preferovaném přístupu. Algoritmus vyhodnocení dotazu při preferenci dotazování B-stromu je uveden v algoritmu 10, zatímco algoritmus 11 představuje vyhodnocení dotazu s preferencí persistentního pole.

4.4.3 Omezení výsledku filtrováním

Po realizaci dotazu do persistentního úložiště, at'už dotazem preferujícím využití B-stromu nebo persistentního pole, je v seznamu nalezených uzlů mnoho těch, které do výsledku nepatří, ale zároveň je nelze rozlišit při prvotním dotazování. Proto je na navrácené výsledky aplikováno filtrování.

Filtrování, reprezentované třídou *Restrictor* a jejími potomky, je využito různě v závislosti na způsobu dotazování, čemuž se věnují odstavce 4.4.3.1 a 4.4.3.2. Poslední odstavec 4.4.3.3 se zabývá popisem jednotlivých tříd filtrování.

Algoritmus 10 Vyhodnocení dotazu preferujícího využití B-stromu

```
Proveď vyhledání v B-stromu dle zástupných identifikátorů
Pro každý nalezený identifikátor:
Pokud je identifikátor odmítnut filtrem:
Pokračuj dalším výsledkem
jinak:
Načti záznam z persistentního pole
Pokud je načtený záznam přijat filtrem:
Ulož záznam do výsledků
Vrať výsledek
```

Algoritmus 11 Vyhodnocení dotazu preferujícího využití persistentního pole

```
Vyhledej v B-stromu kontextový (rodičovský) uzel.
Nastav začátek čtení z persistentního pole dle vyhledaného uzlu
Dokud je splněna ukončovací podmínka:
  Pokud je aktuální záznam přijat filtrem:
Ulož záznam do výsledků
  Načti další záznam
Vrať výsledek
```

Rozhraní třídy *FindRestrictor* poskytuje dva testy, které rozhodují o vyřazení záznamu ze seznamu výsledků. První test umožňuje rozhodnout o splnění podmínky na základě identifikátoru uzlu, což je často možné realizovat ještě před přístupem do persistentního pole. Druhý test je nutný v případě, že první test nerozhodl o platnosti výsledku. Je realizován nad celým záznamem načteným z persistentního pole. Jednotlivé instance potomků třídy *FindRestrictor* je možné kombinovat a tvořit tak složitější podmínky. Kombinace je vždy realizována jako operace AND mezi jednotlivými podmínkami.

4.4.3.1 Použití filtrování při dotazování preferujícím využití B-stromu

Při dotazování preferujícím využití B-stromu je filtr použit na každý ze záznamů, které byly vráceny na dotaz do B-stromu, dvakrát. Nejdříve je vykonána metoda pro test na základě znalosti identifikátoru záznamu. Samotný identifikátor v mnoha případech postačuje k vyhodnocení filtru a algoritmus nemusí načítat celý uzel z persistentního pole. Pouze pokud je podmínka filtru na základě identifikátoru splněna, načte se celý záznam z persistentního pole a vyhodnotí se podmínka filtru na základě celého záznamu. V případě kladného vyhodnocení podmínky je mezivýsledek získaný z dotazu do B-stromu přijat a zařazen do seznamu výsledků.

4.4.3.2 Použití filtrování při dotazování preferujícím využití persistentního pole

Dotazování preferující využití persistentního pole užívá filtrování dvojím způsobem. V průběhu sekvenčního čtení z persistentního pole určuje filtr podmínku, která sekvenční

čtení ukončuje. Jiná instance filtru je poté využita pro ověření, má-li výsledný záznam být zahrnut do seznamu výsledků.

Narozdíl od filtrování dotazů preferujících využití B-stromu je v dotazech preferujících využití persistentního načten vždy celý záznam. Obě podmínky filtru (podmínka dle identifikátoru a dle celého záznamu) jsou tedy vyhodnocovány nad jedním záznamem a nedochází tak k časové úspoře, jako je tomu u filtrování dotazů preferujících využití B-stromu.

4.4.3.3 Popis jednotlivých tříd filtrování

Pro potřebu filtrování jsou implementovány následující třídy *FindRestrictor*:

1. **NullFindRestrictor** - prázdný filtr, který přijme jakýkoli uzel a číslování.
2. **CompositeRestrictor** - filtr, který obsahuje odkaz na jinou instanci filtru. Výsledek je operací AND sebe sama a odkazovaného filtru.
3. **TypeFindRestrictor** - dědí od *CompositeRestrictor* a podmiňuje výsledek dle typu uzlu.
4. **NameFindRestrictor** - dědí od *CompositeRestrictor* a podmiňuje výsledek dle hodnoty uzlu.
5. **EqualityFindRestrictor** - dědí od *CompositeRestrictor* a podmiňuje výsledek dle hodnoty číslování (operace rovná/nerovná se).
6. **DescendantFindRestrictor** - dědí od *CompositeRestrictor* a podmiňuje výsledek na přímé či nepřímé potomky referenčního uzlu.
7. **ValueFindRestrictor** - dědí od *CompositeRestrictor* a podmiňuje výsledek porovnáním textové hodnoty uzlu.

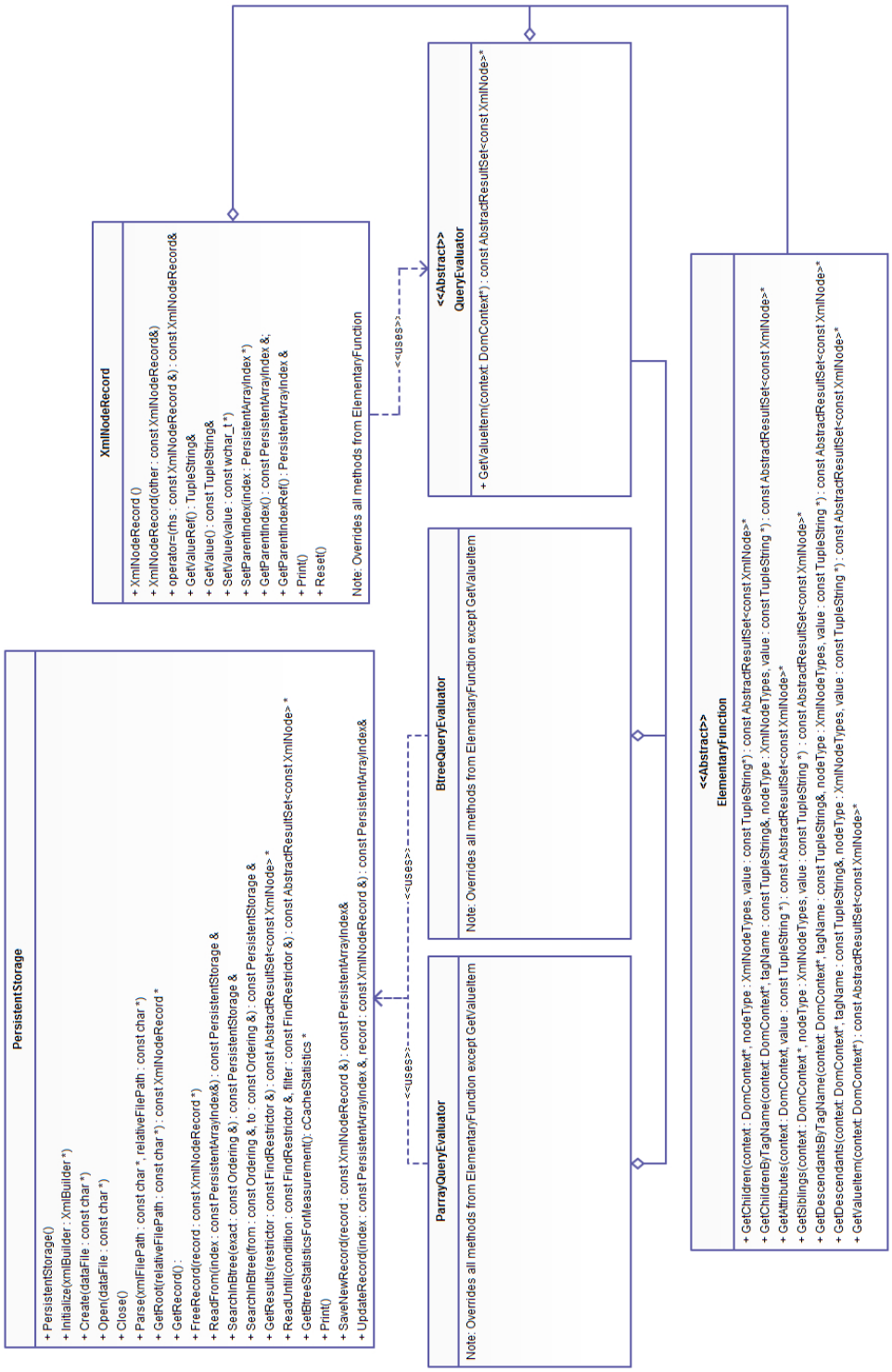
4.4.4 Řešení pro omezení některých číslovacích schémat - odkaz na rodiče uzlu

Některá číslovací schémata neumožňují zjistit, je-li rodič uzlu přímý či nepřímý. Toto přináší problém při porovnání, je-li daný uzel přímým potomkem uzlu referenčního.

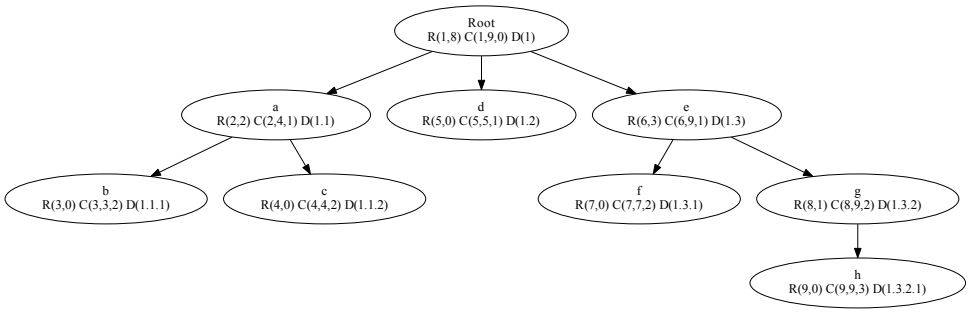
Druhou, často nepodporovanou funkčností je vytvoření zástupného identifikátoru přímého rodiče. Tato funkčnost je třeba zejména v situacích, kdy jsou v B-stromu vyhledáváni sourozenci uzlu, jelikož prvním krokem při této operaci bývá právě vyhledání rodičovského uzlu.

Oba tyto problémy jsou řešeny rozšířením záznamu uzlu *XmlNodeRecord* o ukazatel na rodiče uzlu. To umožňuje obejít popsaný problém a získat umístění rodiče uzlu přímo v persistentním poli.

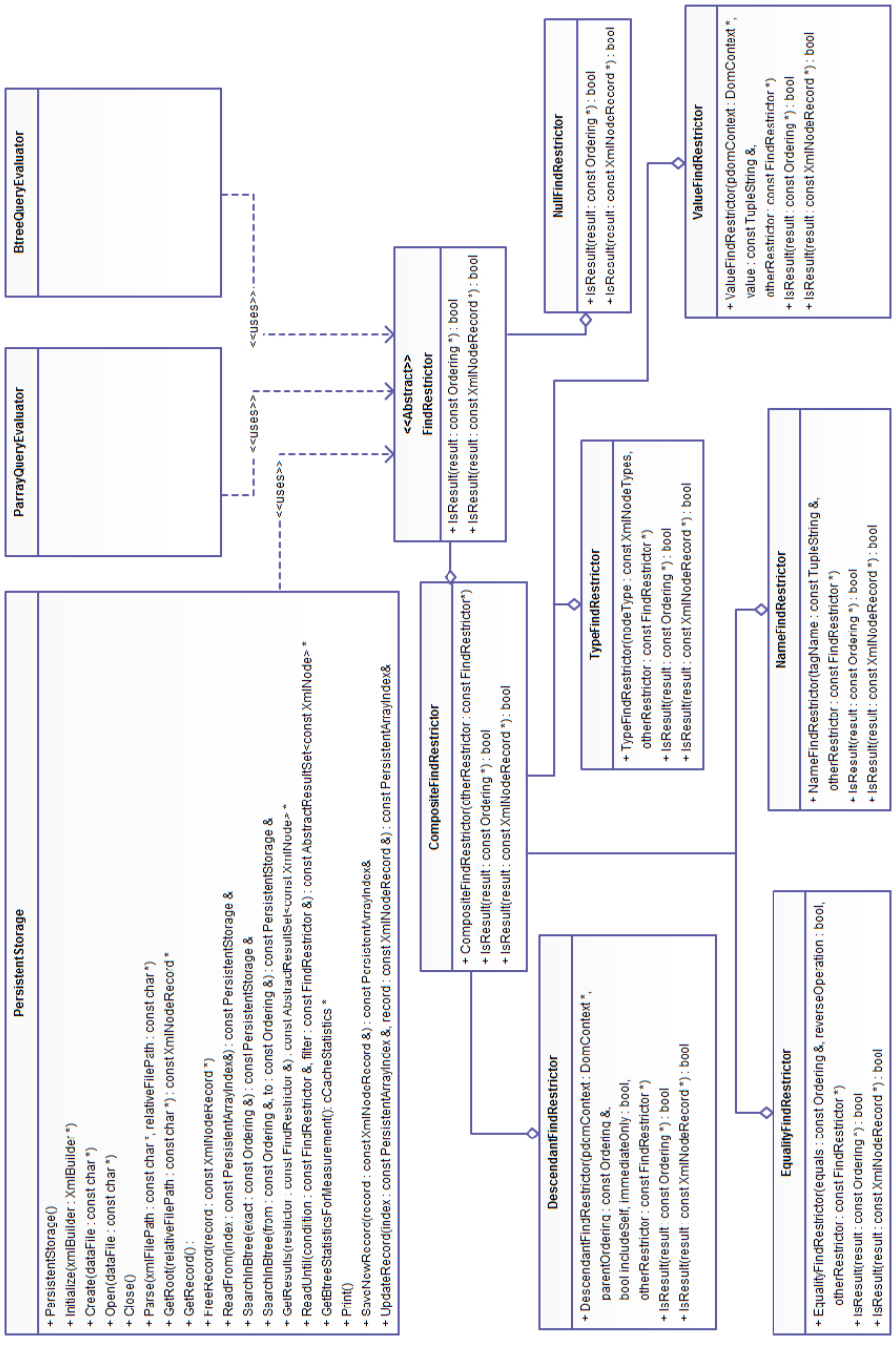
Odkaz na rodiče uzlu je vytvořen a uložen v datových záznamech pouze v případě, že jej použité číslovací schéma ke své činnosti skutečně potřebuje.



Obrázek 8: Třídní diagram třídy QueryEvaluator



Obrázek 9: Očíslovaný XML strom (R značí Range order, C značí Containment order a D značí Dewey order)



Obrázek 10: Diagram tříd dědících od třídy Restrictor, které implementují filtrování

4.5 Omezení popsaného řešení a známé chyby

Obdobně jako u mnoha jiných SW řešení zůstalo v implementovaném řešení několik známých, leč neopravených závad. Vzhledem k tomu, že hlavním přínosem této práce je především praktické prověření výkonnosti zde představené implementace persistentního DOM, nepovažuji odstranění těchto závad za podstatné.

Jedním z omezení této práce je závislost na hloubce zpracovávaného XML souboru. Kvůli vnitřní implementaci B-stromu je nutné při vytváření úložiště persistentního DOM definovat maximální velikost indexovaných identifikátorů číslovacích schémat. To při použití číslovacích schémat *Range order* a *Containment order* není překážkou a vlastní zpracování dokumentu funguje korektně. Problém nastává při použití číslovacího schématu *Dewey order*. Jelikož je toto číslovací schéma závislé na maximální hloubce stromu zpracovávaného XML dokumentu, je při parsování jeho hloubka nejdříve zjištěna. Bohužel pokud je dokument zpracováván do již existujícího úložiště, je tato informace již v úložišti definována a zpracování XML souboru s hloubkou větší než původně definovanou selže. Proto je aplikaci nutné v těchto případech explicitně předat maximální podporovanou hloubku XML souboru. Soubory s hloubkou větší než definovanou při prvním rozparsování do tohoto úložiště však nebude možné úspěšně rozparsovat.

5 Testování a srovnání

Tato kapitola představuje výsledky srovnávacího testování persistentního DOM s implementací klasického DOM. Toto srovnávání má dva hlavní cíle - zhodnotit výkonnost persistentního DOM v závislosti na použitém číslovacím schématu, způsobu vyhodnocení dotazu a porovnání nejvýkonnější varianty s klasickou implementací DOM. K dosažení těchto cílů jsou nejdříve v kapitolách 5.1 a 5.2 popsány testovací sady dat a vlastní proces testování, v kapitole 5.3 je následně představeno srovnání samotného *PDOM* v několika různých konfiguracích a v kapitole 5.4 pak porovnání s klasickým DOM.

Jako reprezentant klasického DOM je použito implementace vytvořené pomocí knihovny Xerces. Pro přehlednější rozlišení mezi persistentním a klasickým DOM v rámci této kapitoly používám zkratky *PDOM* a *XercesDOM*.

5.1 Testovací data

V průběhu srovnávacího testování byla využita testovací data z projektu XMARK a Inex-wiki. Pro testování byly vybrány dvě sady dat, které umožňují otestování různých aspektů práce:

1. Série souborů označená jako **SMALL** je vybrána z inex-wiki a obsahuje 500 souborů malých vzájemně podobných velikostí, které se pohybují v rozmezí 1 - 157 KB.

2. **LARGE** je označení jediného testovacího souboru vytvořeného z projektu XMARK s parametrem 1.0 o velikosti 113MB. Slouží tam, kde nepostačuje srovnání nad malými soubory.

3. **SEQUENTION** je série dat vytvořená z projektu XMARK tak, aby byla velikost souborů lineárně rostoucí. Tato data byla generována s parametry v rozsahu 0.1, 0.2, . . . , 0.6. Velikost souborů začíná na 11MB a končí na 68MB.

5.2 Popis měřených parametrů dotazování

Srovnávací měření je prováděno s cílem určit tendence chování *PDOM* v jednotlivých konfiguracích vůči *XercesDOM*. Velikost měřených dat je proto omezena (největší XML soubor má cca 68 MB), přestože silnou stránkou *PDOM* je právě zpracování souborů značných velikostí. Toto omezení má ryze praktické zdůvodnění - díky omezení velikosti testovací sady bylo možné provést kvalitnější automatické testování, kdy jsou v *XercesDOM* a *PDOM* testovány podobné dotazy.

5.2.1 Měřené vlastnosti

V průběhu testování *PDOM* jsou sledovány následující vlastnosti:

1. čas, za kterou je operace vykonána,

2. počet přístupů k datovému úložišti potřebný pro vykonání operace,

Číslo konfigurace	Číslovací schéma	Způsob vyhodnocení dotazu
1	Containment order	BTREE usage preference
2	Containment order	PARRAY usage preference
3	Dewey order	BTREE usage preference
4	Dewey order	PARRAY usage preference
5	Range order	BTREE usage preference
6	Range order	PARRAY usage preference

Tabulka 2: Přehled konfigurací PDOM

3. množství alokované paměti,
4. velikost vstupního souboru,
5. velikost datového souboru,
6. maximální hloubka xml souboru.

Vzhledem k povaze klasického *XercesDOM* jsou pro něj měřeny pouze následující vlastnosti:

1. čas, za kterou je operace vykonána,
2. množství alokované paměti,
3. velikost vstupního souboru.

Ostatní vlastnosti měřené u *PDOM* jsou pro *XercesDOM* nesmyslné, především z důvodu rozdílné povahy *DOM*.

Maximální hloubka XML souboru není pro *XercesDOM* měřena, jelikož narozdíl od *PDOM* nezávisí jeho implementace na této vlastnosti.

5.2.2 Parametry dotazování a postup jeho vyhodnocení

Pod pojmem **konfigurace PDOM** je v průběhu měření rozeznávána šestice nastavení, která označuje jednotlivé možnosti vyhodnocení dotazování (viz tabulka 2). Tato šestice je tvořena kombinací dvou způsobů vyhodnocení dotazu a tří podporovaných číslovacích schémat. V jednotlivých grafech a tabulkách jsou pro přehlednost číslovací schémata označena jako *Dewey order*, *Range order* a *Containment order*, zatímco způsob vyhodnocení dotazu jako *BTREE usage preference* pro vyhodnocení preferující využití B-stromu a *PARRAY usage preference* pro vyhodnocení preferující využití persistentního pole.

Při testování dotazování persistetního DOM jsou testovány vždy všechny operace, tedy *getDescendants*, *getDescendantsByTagName*, *getChildren*, *getChildrenByTagName*, *getAttributes*, *getSiblings* a *getValue*. Pro přehlednost prezentovaných výsledků je pro některé z grafů vybrána pouze jedna z uvedených operací.

Postup testování se řídí algoritmem popsaným v algoritmu 12. V rámci tohoto algoritmu je provedeno vyhodnocení dotazu nejdříve pro *PDOM* a následně pro *xercesDOM*

na každou z operací a v případě *PDOM* pro každou z podporovaných konfigurací. Dotazování je vždy realizováno nad uzlem, pro který je danou operací vrácen neprázdný výsledek. Během jednotlivých iterací pro konkrétní konfiguraci jsou náhodně zvolené uzly cachovány. To zajišťuji použitím identických vstupních argumentů pro každou z operací v dané konfiguraci.

Algoritmus 12 Pseudokód popisující algoritmus testování *PDOM* a *XercesDOM*.

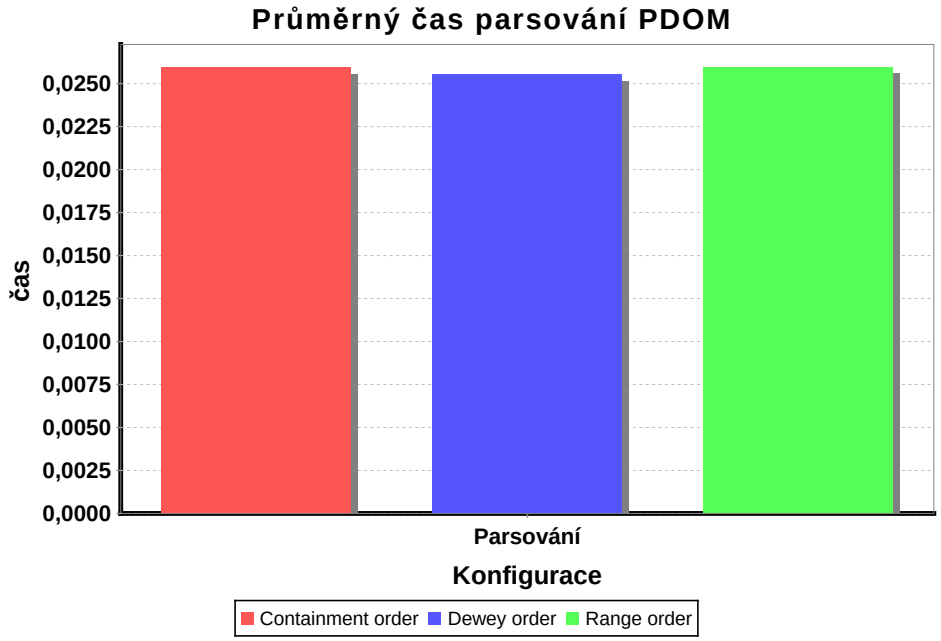
```
Rozparsování všech vstupních souborů do datových struktur
Pro každou variantu konfigurace PDOM:
  Do X ulož výsledek neměřeného dotazu getChildren
  Prokaždou operaci O:
Vyber z X vhodný uzel do Y
Pokud je Y prázdné:
  Do X ulož výsledek neměřeného dotazu getDescendants
  Vyber z X vhodný uzel do Y
  Realizuj měřený dotaz operace
O Pro XercesDOM:
  Do X ulož výsledek neměřeného dotazu getChildren
  Prokaždou operaci O:
Vyber z X vhodný uzel do Y
Pokud je Y prázdné:
  Do X ulož výsledek neměřeného dotazu getDescendants
  Vyber z X vhodný uzel do Y
  Realizuj měřený dotaz operace O
```

5.3 Zhodnocení výkonnosti PDOM implementace v závislosti na použité konfiguraci

První sledovanou vlastností je průměrný čas, který je potřeba k dokončení jednotlivých operací nad *PDOM*/*DOM*. Pro přehlednost se dělí do dvou částí, z nichž první srovnává čas potřebný k rozparsování vstupního XML do datových struktur a druhá srovnává čas pro vykonání jednotlivých operací.

Průměrný čas parsování dat do datového souboru je zobrazen v grafu na obrázku 11 a tabulce 3. Tento graf je vytvořen nad testovací sadou *SMALL*. Naměřené hodnoty jsou pro jednotlivá číslovací schémata naprosto srovnatelné. Vzhledem k nejednoznačnosti naměřených výsledků graf 12 a tabulka 4 prezentují data i nad datovou sadou *LARGE*. V datové sadě *LARGE* je *Dewey order* při zpracování o něco rychlejší než *Range* a *Containment order*. To lze přičíst na vrub nutnosti aktualizace záznamů u druhých dvou číslovacích schémat, zatímco *Dewey order* je vkládán přímo ve finální podobě.

Průměrný čas vyhodnocení dotazu je prezentován v grafu 13 a v tabulce 5. Ze zobrazených vyhodnocení dotazů je patrná vyšší časová náročnost vyhodnocení některých



Obrázek 11: Průměrný čas (ve vteřinách) parsování vstupního souboru pro *PDOM* pro datovou sadu *SMALL*

Číslovací schéma	**Průměr**
Containment order	25.949
Dewey order	25.550
Range order	25.978

Tabulka 3: Naměřené hodnoty v ms pro parsování *PDOM* nad datovou sadou *SMALL*

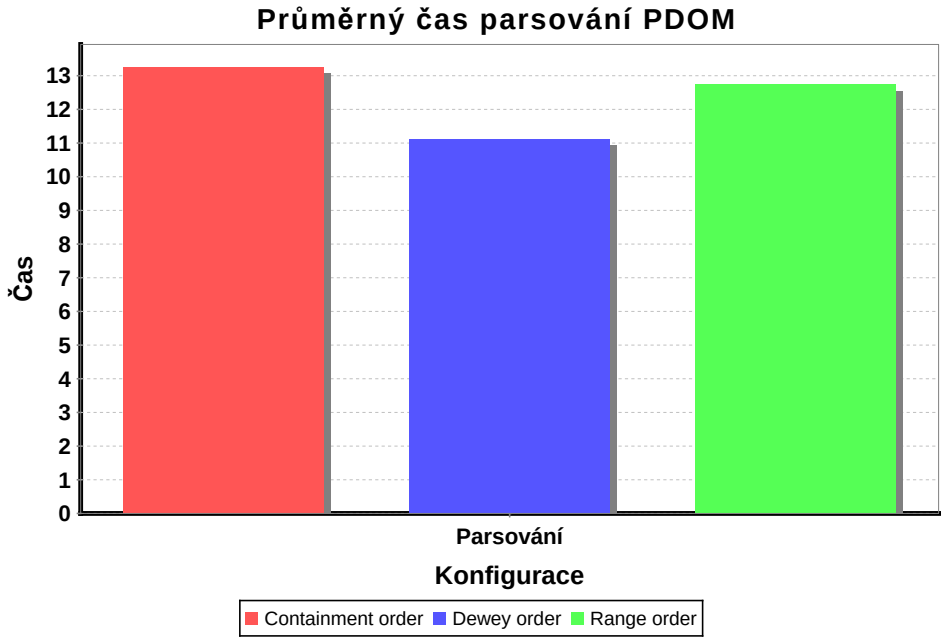
operací pro *Range order*. Tuto vyšší časovou náročnost lze přičíst na vrub nutnosti ověřování, je-li dotazovaný uzel přímým potomkem či nikoli.

Dewey a *Containment order* totiž umožňují zjistit, je-li uzel přímým potomkem uzlu jiného pouze na základě samotného identifikátoru číslovacího schématu. To je zpravidla možné provést bez nutnosti načtení celého záznamu z persistetního pole. Na druhou stranu *Range order* tuto informaci neuchovává a pro její vyhodnocení je třeba načtení celého záznamu z persistentního pole.

Pro úplnost je v grafu 14 a tabulce 6 prezentováno i srovnání nad datovou sadou *LARGE*, které vykazuje totožné charakteristiky jako testy nad datovou sadou *SMALL*.

Číslovací schéma	**Průměr**
Containment order	13265
Dewey order	11123
Range order	12749

Tabulka 4: Naměřené hodnoty v ms pro parsování *PDOM* pro datovou sadu *LARGE*



Obrázek 12: Čas parsování vstupního souboru (ve vteřinách) pro *PDOM* pro datovou sadu *LARGE*

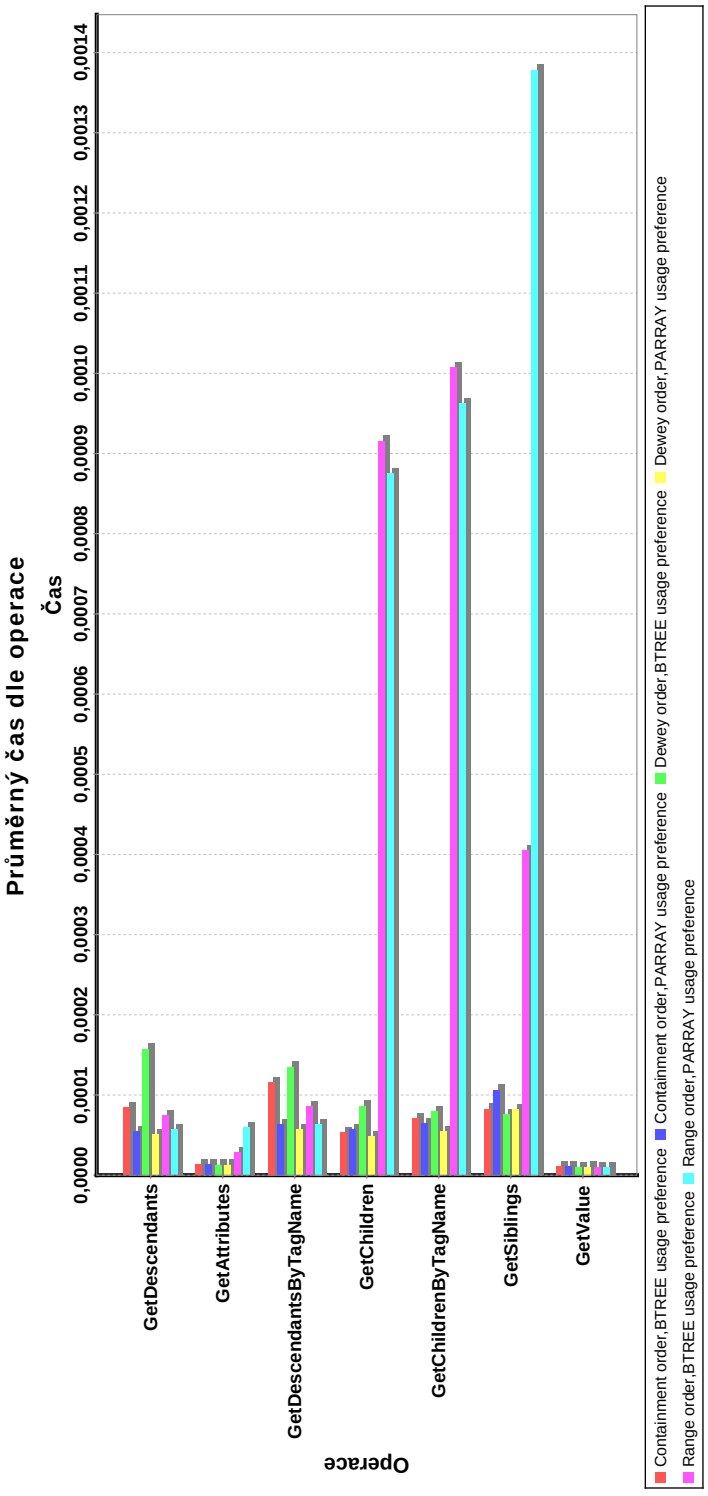
Jediným rozdílem je drobný nárůst v časové náročnosti vyhodnocení dotazu, který lze však s přihlédnutím k větší velikosti vstupního souboru očekávat.

Název operace	Číslovací schéma	Preference vyhodnocení	Průměr
GetDescendants	Containment order	B-strom	0.085
GetDescendants	Dewey order	B-strom	0.157
GetDescendants	Range order	B-strom	0.075
GetAttributes	Containment order	B-strom	0.013
GetAttributes	Dewey order	B-strom	0.013
GetAttributes	Range order	B-strom	0.028
GetDescendantsByTagName	Containment order	B-strom	0.116
GetDescendantsByTagName	Dewey order	B-strom	0.135
GetDescendantsByTagName	Range order	B-strom	0.086
GetChildren	Containment order	B-strom	0.052
GetChildren	Dewey order	B-strom	0.086
GetChildren	Range order	B-strom	0.915
GetChildrenByTagName	Containment order	B-strom	0.071
GetChildrenByTagName	Dewey order	B-strom	0.079
GetChildrenByTagName	Range order	B-strom	1.007
GetSiblings	Containment order	B-strom	0.082
GetSiblings	Dewey order	B-strom	0.075
GetSiblings	Range order	B-strom	0.405
GetValueItem	Containment order	B-strom	0.010
GetValueItem	Dewey order	B-strom	0.009
GetValueItem	Range order	B-strom	0.009
GetDescendants	Containment order	persistnentní pole	0.054
GetDescendants	Dewey order	persistnentní pole	0.050
GetDescendants	Range order	persistnentní pole	0.057
GetAttributes	Containment order	persistnentní pole	0.012
GetAttributes	Dewey order	persistnentní pole	0.012
GetAttributes	Range order	persistnentní pole	0.059
GetDescendantsByTagName	Containment order	persistnentní pole	0.063
GetDescendantsByTagName	Dewey order	persistnentní pole	0.056
GetDescendantsByTagName	Range order	persistnentní pole	0.063
GetChildren	Containment order	persistnentní pole	0.057
GetChildren	Dewey order	persistnentní pole	0.047
GetChildren	Range order	persistnentní pole	0.875
GetChildrenByTagName	Containment order	persistnentní pole	0.064
GetChildrenByTagName	Dewey order	persistnentní pole	0.054
GetChildrenByTagName	Range order	persistnentní pole	0.962
GetSiblings	Containment order	persistnentní pole	0.106
GetSiblings	Dewey order	persistnentní pole	0.081
GetSiblings	Range order	persistnentní pole	1.378
GetValueItem	Containment order	persistnentní pole	0.010
GetValueItem	Dewey order	persistnentní pole	0.010
GetValueItem	Range order	persistnentní pole	0.009

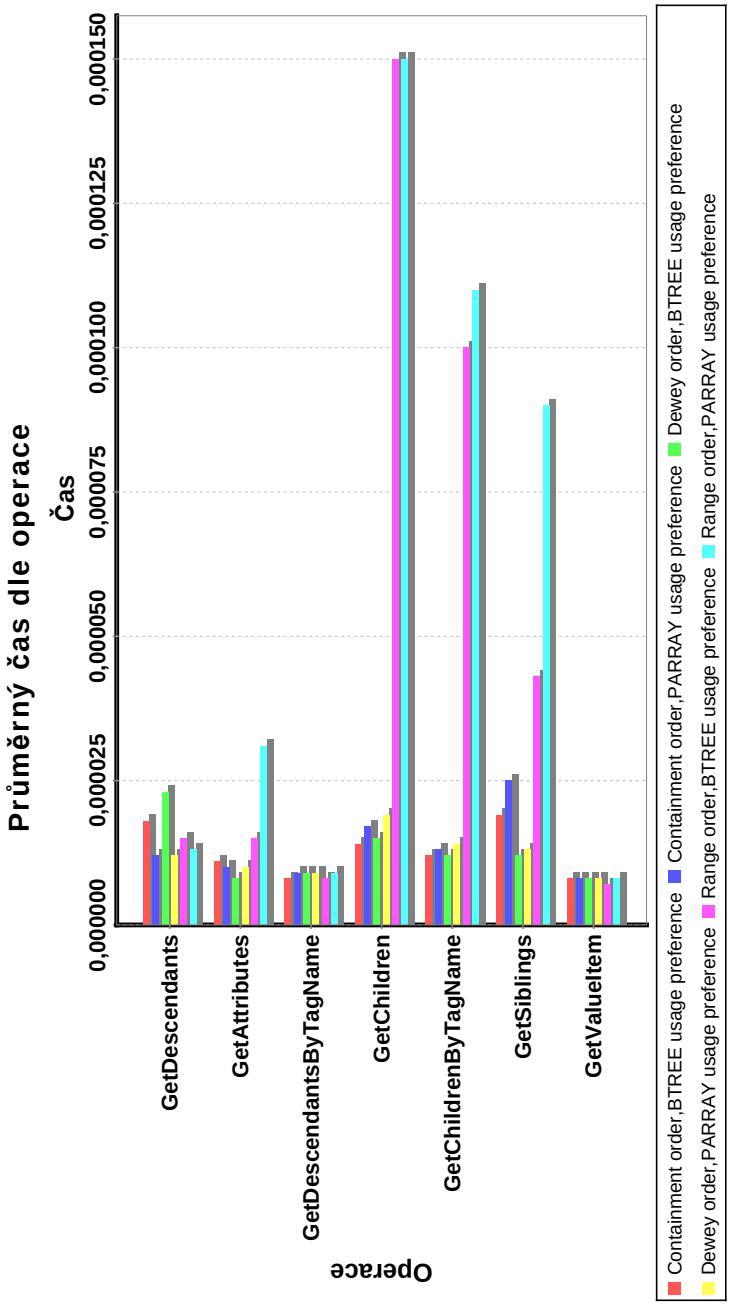
Tabulka 5: Čas dotazování v ms jednotlivých operací *PDOM* pro datovou sadu *SMALL*

Název operace	Číslovací schéma	Preference vyhodnocení	Průměr
GetDescendants	Containment order	B-strom	0.018
GetDescendants	Dewey order	B-strom	0.023
GetDescendants	Range order	B-strom	0.015
GetAttributes	Containment order	B-strom	0.011
GetAttributes	Dewey order	B-strom	0.008
GetAttributes	Range order	B-strom	0.015
GetDescendantsByTagName	Containment order	B-strom	0.008
GetDescendantsByTagName	Dewey order	B-strom	0.009
GetDescendantsByTagName	Range order	B-strom	0.008
GetChildren	Containment order	B-strom	0.014
GetChildren	Dewey order	B-strom	0.015
GetChildren	Range order	B-strom	0.149
GetChildrenByTagName	Containment order	B-strom	0.012
GetChildrenByTagName	Dewey order	B-strom	0.012
GetChildrenByTagName	Range order	B-strom	0.100
GetSiblings	Containment order	B-strom	0.019
GetSiblings	Dewey order	B-strom	0.012
GetSiblings	Range order	B-strom	0.043
GetValueItem	Containment order	B-strom	0.008
GetValueItem	Dewey order	B-strom	0.008
GetValueItem	Range order	B-strom	0.007
GetDescendants	Containment order	persistnentní pole	0.012
GetDescendants	Dewey order	persistnentní pole	0.012
GetDescendants	Range order	persistnentní pole	0.013
GetAttributes	Containment order	persistnentní pole	0.010
GetAttributes	Dewey order	persistnentní pole	0.010
GetAttributes	Range order	persistnentní pole	0.031
GetDescendantsByTagName	Containment order	persistnentní pole	0.009
GetDescendantsByTagName	Dewey order	persistnentní pole	0.009
GetDescendantsByTagName	Range order	persistnentní pole	0.009
GetChildren	Containment order	persistnentní pole	0.017
GetChildren	Dewey order	persistnentní pole	0.019
GetChildren	Range order	persistnentní pole	0.150
GetChildrenByTagName	Containment order	persistnentní pole	0.013
GetChildrenByTagName	Dewey order	persistnentní pole	0.014
GetChildrenByTagName	Range order	persistnentní pole	0.110
GetSiblings	Containment order	persistnentní pole	0.025
GetSiblings	Dewey order	persistnentní pole	0.013
GetSiblings	Range order	persistnentní pole	0.090
GetValueItem	Containment order	persistnentní pole	0.008
GetValueItem	Dewey order	persistnentní pole	0.008
GetValueItem	Range order	persistnentní pole	0.008

Tabulka 6: Čas dotazování v ms jednotlivých operací *PDOM* pro datovou sadu *LARGE*

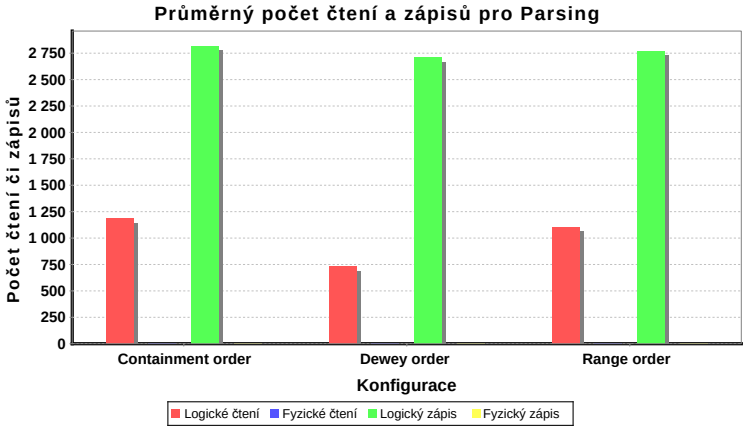


Obrázek 13: Čas dotazování jednotlivých operací (ve vteřinách) *PDOM* pro datovou sadu *SMALL*



Obrázek 14: Čas dotazování jednotlivých operací (ve vteřinách) *PDOM* pro datovou sadu *LARGE*

Vhodné doplnění informací o náročnosti dotazování umožňuje srovnání vstupně výstupních operací. V grafu 15 a tabulce 7 je znázorněno srovnání těchto operací pro operaci parsování. Z grafu je patrné, že v průběhu parsování dochází jak k zápisu tak i ke čtení, přičemž množství čtení je nejnižší pro číslovací schéma *Dewey order*. To souvisí s nutností provádět aktualizace již uložených záznamů u číslovacích schémat *Containment* a *Range order*. Množství zápisů i čtení je mezi jednotlivými číslovacími schématy srovnatelné, což odpovídá srovnatelné době parsování.



Obrázek 15: Průměrný počet čtení a zápisů pro parsování

V grafu 16 je znázorněn průměrný počet čtení a zápisů pro operaci *getAttributes*, jež byla zvolena pro reprezentaci efektivity dotazování. Tabulka 8 zobrazuje počty čtení pro všechny operace rozdělené dle způsobu vyhodnocení dotazování. Z grafu je patrné, že jsou pro tuto operaci jednotlivé konfigurace víceméně srovnatelné, možná s mírnou výhodností pro *Containment order* a *Dewey order* při použití vyhodnocování s preferencí persistentního pole.

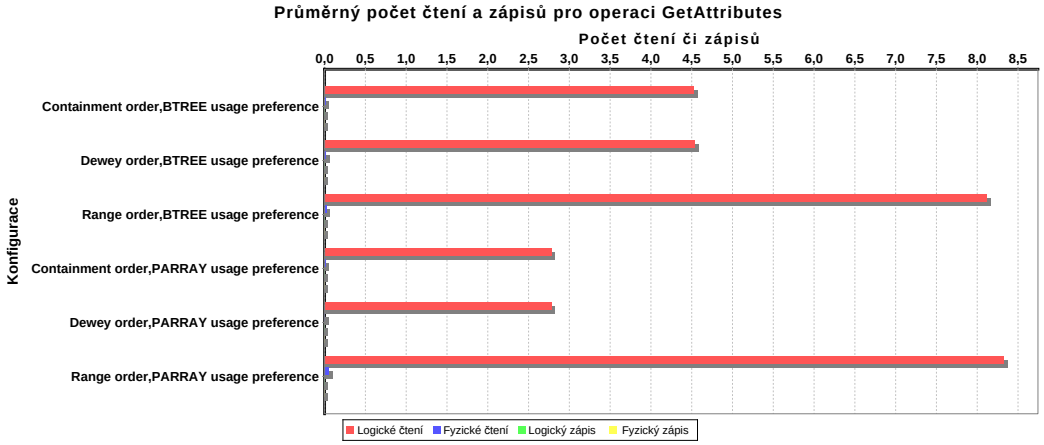
Je však nutné si uvědomit, že metoda *getAttributes* vrací zpravidla velmi malé množství elementů, často dokonce pouze jeden. Neefektivita některých konfigurací se tak nemusí projevit. Z tohoto důvodu je v grafu 17 prezentován i počet čtení a zápisů pro operaci *getDescendants*, která zpravidla vrací velký počet výsledků. Lze tak porovnat, jaké množství přístupů k diskovému úložišti je třeba pro vyhodnocení náročnější operace.

Z uvedeného grafu je patrné, že zatímco použité číslovací schéma je méně důležitou vlastností, způsobem vyhodnocení dotazu lze docílit výrazného zmenšení počtu přístupů na disk u dotazů, které typicky vracejí větší množství uzlů.

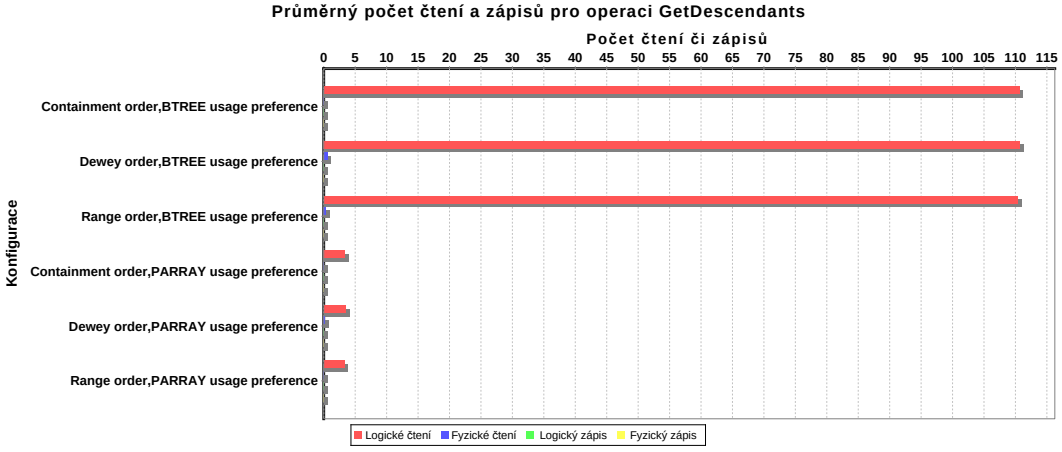
Číslovací schéma	Logické čtení	Fyzické čtení	Logický zápis	Fyzický zápis
Containment order	1183	0	2818	0
Dewey order	729	0	2708	0
Range order	1104	0	2770	0

Tabulka 7: Počet čtení a zápisů pro Parsování

Při podrobném prozkoumání přiložených tabulek je znatelné zvýšené množství čtení pro operace *getChildren*, *getChildrenByTagName*, *getSiblings* a méně zřetelně i u operace *getAttributes* u číslovacího schématu *Range order*. Toto zvýšené množství přístupů je příčinou delšího času vykonání zmíněných operací, jenž byl již v této kapitole zmíněn. Je způsoben nutností ověřování, je-li dotazovaný uzel přímým potomkem či nikoli, která plyne z vlastností tohoto číslovacího schématu.



Obrázek 16: Průměrný počet čtení a zápisů pro operaci getAttributes



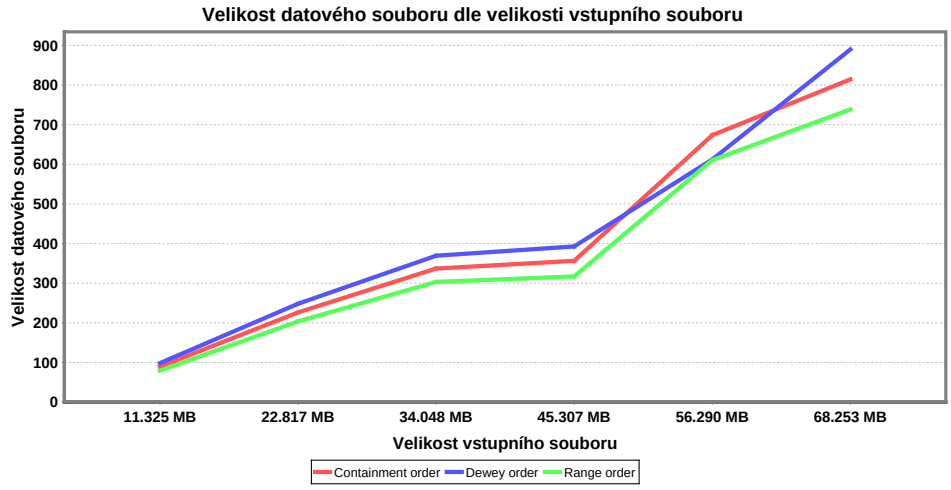
Obrázek 17: Průměrný počet čtení a zápisů pro operaci getDescendants

Operace	Číslovací schéma	Preference vyhodnocení	Logické čtení
GetDescendants	Containment order	B-strom	110
GetDescendants	Dewey order	B-strom	110
GetDescendants	Range order	B-strom	110
GetAttributes	Containment order	B-strom	4
GetAttributes	Dewey order	B-strom	4
GetAttributes	Range order	B-strom	8
GetDescendantsByTagName	Containment order	B-strom	131
GetDescendantsByTagName	Dewey order	B-strom	132
GetDescendantsByTagName	Range order	B-strom	131
GetChildren	Containment order	B-strom	10
GetChildren	Dewey order	B-strom	10
GetChildren	Range order	B-strom	267
GetChildrenByTagName	Containment order	B-strom	9
GetChildrenByTagName	Dewey order	B-strom	9
GetChildrenByTagName	Range order	B-strom	227
GetSiblings	Containment order	B-strom	24
GetSiblings	Dewey order	B-strom	23
GetSiblings	Range order	B-strom	214
GetValueItem	Containment order	B-strom	2
GetValueItem	Dewey order	B-strom	2
GetValueItem	Range order	B-strom	2
GetDescendants	Containment order	persistnentní pole	3
GetDescendants	Dewey order	persistnentní pole	3
GetDescendants	Range order	persistnentní pole	3
GetAttributes	Containment order	persistnentní pole	2
GetAttributes	Dewey order	persistnentní pole	2
GetAttributes	Range order	persistnentní pole	8
GetDescendantsByTagName	Containment order	persistnentní pole	3
GetDescendantsByTagName	Dewey order	persistnentní pole	3
GetDescendantsByTagName	Range order	persistnentní pole	3
GetChildren	Containment order	persistnentní pole	3
GetChildren	Dewey order	persistnentní pole	3
GetChildren	Range order	persistnentní pole	137
GetChildrenByTagName	Containment order	persistnentní pole	3
GetChildrenByTagName	Dewey order	persistnentní pole	3
GetChildrenByTagName	Range order	persistnentní pole	117
GetSiblings	Containment order	persistnentní pole	2
GetSiblings	Dewey order	persistnentní pole	3
GetSiblings	Range order	persistnentní pole	175
GetValueItem	Containment order	persistnentní pole	2
GetValueItem	Dewey order	persistnentní pole	2
GetValueItem	Range order	persistnentní pole	2

Tabulka 8: Počet čtení pro jednotlivé způsoby dotazování

Poslední srovnávanou veličinou u srovnání konfigurací *PDOM* je velikost datového souboru. Měření bylo provedeno na testovacích datech označených *SEQUENTION*. V grafu 18 a tabulce 9 lze pozorovat vývoj velikosti datového souboru dle velikosti vstupního souboru pro použitá číslovací schémata. Z grafu je patrné, že zatímco *Range order* má identifikátor tvořený dvěma číslicemi (64 bajtů na testované platformě), *Containment order* číslicemi třemi (96 bajtů). Narozdíl od těchto číslovacích schémat není velikost identifikátoru *Dewey order* konstantní, nýbrž závislá na hloubce zanoření uzlu. V grafu tedy její křivka nekopíruje přesně křivku ostatních číslicových schémat.

Při podrobném prozkoumání lze dojít k závěru, že se křivky *Range order* a *Dewey order* (méně zřetelně i *Containment order* a *Dewey order*) rozbíhají podezřele málo oproti uvedeným výpočtům velikosti jejich identifikátorů. Vždyť testované XML soubory by musely mít průměrnou hloubku zanoření všech svých uzlů jen o málo větší než 2. To je při maximální hloubce souborů 13 (viz tabulka 9) velmi nepravděpodobné. Důvodem této situace je nutnost uchovávat odkaz na rodičovský uzel u číslovacích schémat *Range order* a *Containment order*, což v případě číslovacího schématu *Range order* efektivně zdvojnásobuje množství paměti potřebné k uložení záznamu o uzlu.



Obrázek 18: Velikost datového souboru v závislosti na velikosti souboru vstupního

Z testů uvedených v této kapitole vyplývá několik zajímavých poznatků o nejefektivnější konfiguraci *PDOM*. Zatímco pro jednoduché dotazy, které vracejí malé množství elementů (typicky se jedná o operace jako *getValue* či *getAttributes*), nemá konfigurace velký vliv na rychlost vyhodnocení, u složitějších dotazů (například *getDescendants* či *getChildren*) je situace jasnější.

Největší úsporou v počtu přístupů k disku, která se na většině běžných počítačů projeví výraznou časovou úsporou, je volba vyhodnocení dotazu s preferencí persistentního pole. Přestože v grafech srovnávajících čas vykonání jednotlivých operací není vyhodnocení dotazu s preferencí persistentního pole jednoznačně efektivnější než vyhodnocení dotazu s preferencí B-stromu, považuje autor při přihlédnutí ke srovnání množství přístupů na disk vyhodnocení s preferencí persistentního pole za efektivnější než vyhodnocení

Název souboru	Číslovací schéma	Max. hloubka	Velikost XML souboru	Velikost datového souboru
output-0.1.xml	Containment order	13	11.325 MB	89.305 MB
output-0.1.xml	Dewey order	13	11.325 MB	98.375 MB
output-0.1.xml	Range order	13	11.325 MB	79.422 MB
output-0.2.xml	Containment order	13	22.817 MB	226.258 MB
output-0.2.xml	Dewey order	13	22.817 MB	248.164 MB
output-0.2.xml	Range order	13	22.817 MB	203.609 MB
output-0.3.xml	Containment order	13	34.048 MB	337.055 MB
output-0.3.xml	Dewey order	13	34.048 MB	369.383 MB
output-0.3.xml	Range order	13	34.048 MB	303.258 MB
output-0.4.xml	Containment order	13	45.307 MB	356.484 MB
output-0.4.xml	Dewey order	13	45.307 MB	392.766 MB
output-0.4.xml	Range order	13	45.307 MB	317.148 MB
output-0.5.xml	Containment order	13	56.290 MB	673.445 MB
output-0.5.xml	Dewey order	13	56.290 MB	612.234 MB
output-0.5.xml	Range order	13	56.290 MB	610.625 MB
output-0.6.xml	Containment order	13	68.253 MB	814.313 MB
output-0.6.xml	Dewey order	13	68.253 MB	889.750 MB
output-0.6.xml	Range order	13	68.253 MB	738.398 MB

Tabulka 9: Velikost datového souboru v závislosti na velikost XML souboru

s preferencí přístupu k B-stromu. K nejednoznačnému srovnání v časové složitosti dochází kvůli štědrému nastavení cachování přístupu k disku a v případě náročnějších operací se vyhodnocení s využitím persistentního pole projevuje jako efektivnější.

Dalším významným omezením v rychlosti vykonání dotazu je použití číslovacího schématu *Range order*. Ten vzhledem k nemožnosti rozlišení mezi přímým a nepřímým rodičem výrazně zpomaluje vyhodnocení operací, kde je tato znalost vyžadována.

Poměrně překvapivě skončilo srovnání číslovacích schémat *Containment order* a *Dewey order*. Přestože *Containment order* neumožňuje vytvoření zástupného identifikátoru rodičovského elementu, což vyžaduje pro některé typy dotazů (typicky operace *getSiblings*) nutnost uchování explicitního odkazu na rodičovský uzel v záznam o uzlu, není toto číslovací schéma pomalejší než číslovací schéma *Dewey order*, jež vytvoření zástupného identifikátoru umožňuje.

S přihlédnutím k nárokům na velikost datového souboru v závislosti na použitém číslovacím schématu lze za nejvýhodnější číslovací schéma celkem jednoznačně prohlásit *Containment order*.

5.4 Porovnání XercesDOM a PDOM

Při porovnání *PDOM* a *XercesDOM* je *PDOM* reprezentováno nejefektivnější konfigurací zjištěnou v předchozí kapitole - číslovacím schématem *Containment order* a vyhodnocením dotazu s preferencí persistentního pole (*PARRAY*).

První porovnávanou veličinou je doba parsování dokumentu, která je uvedena v grafu 19 a tabulce 10. Testovací běh byl vykonán nad datovou sadou *SMALL*. Z grafu je zřejmé, že parsování dokumentu je výrazně rychlejší pro *XercesDOM*, zejména kvůli přístupu k pomalému diskovému úložišti, který musí *PDOM* narozdíl od v paměti pracujícího

Typ DOM	Průměr
PDOM	22.636
Xerces	0.884

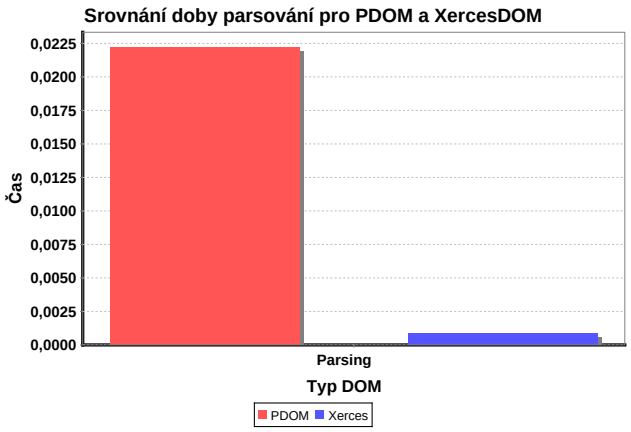
Tabulka 10: Srovnání průměrné doby parsování v ms pro *XercesDOM* a *PDOM* pro datovou sadu *SMALL*

Typ DOM	Průměr
PDOM	13265.000
Xerces	2276.000

Tabulka 11: Srovnání průměrné doby parsování v ms pro *XercesDOM* a *PDOM* pro datovou sadu *LARGE*

XercesDOM vykonávat. Je však nutné podotknout, že zatímco *XercesDOM* musí udržovat rozparsovanou podobu dokumentu v paměti a tedy relativně často parsování opakovat, může *PDOM* provést rozparsování jednou na pevné úložiště a následně dlouhodobě využívat již rozparsovaných dokumentů. Obdobně je třeba u *XercesDOM* brát v potaz maximální množství paměti, která je k dispozici operačnímu systému.

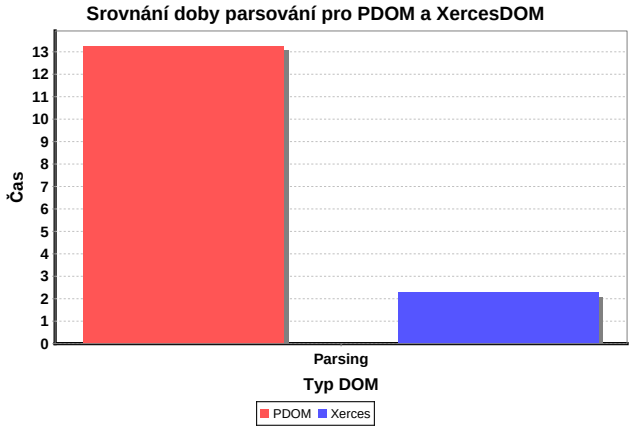
Pro možnost srovnání je v grafu 20 a tabulce 11 uvedena doba parsování nad datovou sadou *LARGE*. Ze srovnání je patrné, že přestože je *XercesDOM* stále efektivnější v parsování vstupního XML souboru, roste jeho časová náročnost parsování rychleji než v případě *PDOM*.



Obrázek 19: Srovnání průměrné doby parsování (ve vteřinách) pro *XercesDOM* a *PDOM* pro datovou sadu *SMALL*

Další důležitou srovnávanou veličinou je porovnání časové efektivity dotazování. To bylo realizováno jak nad malými soubory (datová sada *SMALL*), tak i velkými soubory (datová sada *LARGE*).

Srovnání průměrné doby dotazování nad datovou sadou *SMALL* lze nalézt v grafu na obrázku 21 a v tabulce 12. Z grafu je patrné, že pro malé soubory je mírně výkonnější



Obrázek 20: Srovnání průměrné doby parsování (ve vteřinách) pro *XercesDOM* a *PDOM* pro datovou sadu *LARGE*

XercesDOM. *PDOM* nicméně získává na výkonnosti v situacích, kdy je typicky zpracováváno větší množství uzlů (operace *getDescendants*).

V grafu na obrázku 22 a tabulce 13 vidíme vykonání stejných operací jako v případě předchozím, jen nad datovou sadou *LARGE*. V těchto tabulkách je zřetelné, že pro *PDOM* nedochází k degradaci rychlosti vyhodnocení dotazů na základě velikosti vstupního souboru ani kvůli většímu množství nalezených výsledků.

Pro *XercesDOM* je situace poněkud jiná. Z uvedeného grafu je zřetelná degradace rychlosti vyhodnocení dotazů, přestože jsou všechny operace realizovány v paměti počítače. Markantní zpomalení je u operací *getChildren* a *getChildrenByTagName*, méně významné poté u *getDescendants* a *getDescendantsByTagName*. Způsobuje ho neoptimálním zacházením z uzly v implementaci *XercesDOM*, při kterém dochází k alokaci dat, a souvisejícím zpomalením výpočtu při větším množství nalezených uzlů.

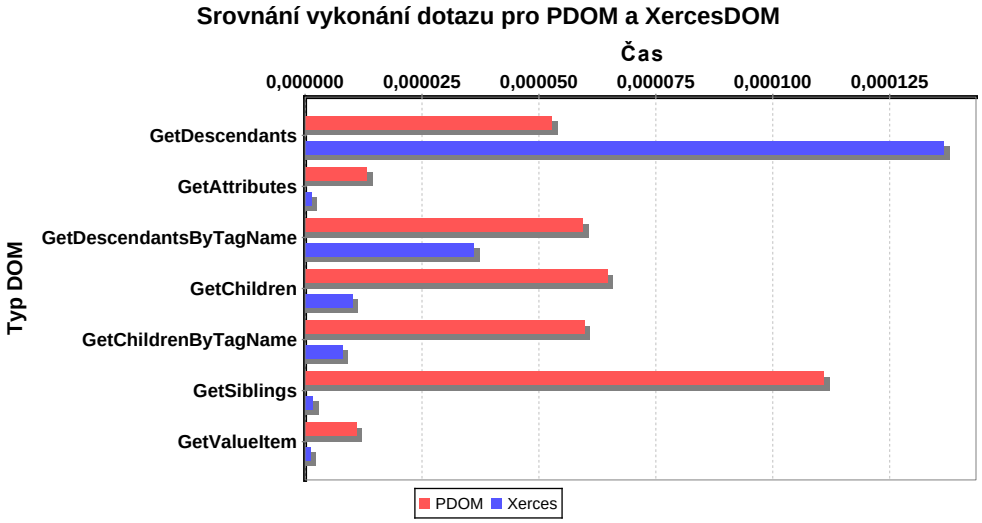
Metody *getAttributes*, *getSiblins* a *getValue* napovídají, že by k degradaci rychlosti u *XercesDOM* docházet nemuselo, není však dostatečně průkazné pro vyslovení jednoznačného závěru.

Název operace	Typ implementace DOM	**Průměr**
GetDescendants	PDOM	0.052
GetAttributes	PDOM	0.013
GetDescendantsByTagName	PDOM	0.059
GetChildren	PDOM	0.064
GetChildrenByTagName	PDOM	0.059
GetSiblings	PDOM	0.110
GetValueItem	PDOM	0.010
GetDescendants	XercesDOM	0.136
GetAttributes	XercesDOM	0.001
GetDescendantsByTagName	XercesDOM	0.036
GetChildren	XercesDOM	0.010
GetChildrenByTagName	XercesDOM	0.007
GetSiblings	XercesDOM	0.001
GetValueItem	XercesDOM	0.001

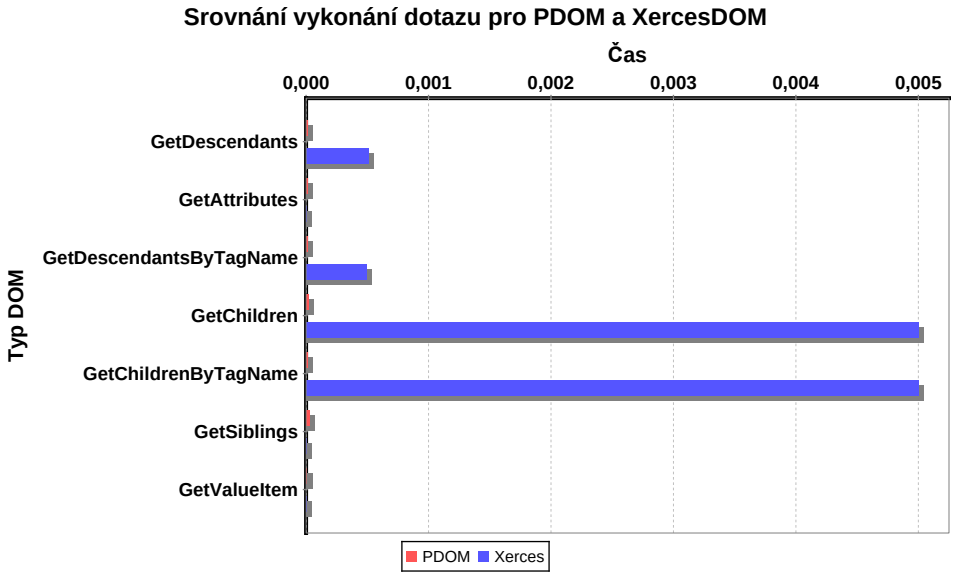
Tabulka 12: Srovnání průměrné doby dotazování v ms jednotlivých operací nad datovou sadou *SMALL*

Název operace	Typ implementace DOM	**Průměr**
GetDescendants	PDOM	0.012
GetAttributes	PDOM	0.0100
GetDescendantsByTagName	PDOM	0.009
GetChildren	PDOM	0.017
GetChildrenByTagName	PDOM	0.013
GetSiblings	PDOM	0.025
GetValueItem	PDOM	0.008
GetDescendants	XercesDOM	0.509
GetAttributes	XercesDOM	0.000
GetDescendantsByTagName	XercesDOM	0.490
GetChildren	XercesDOM	5.000
GetChildrenByTagName	XercesDOM	5.000
GetSiblings	XercesDOM	0.002
GetValueItem	XercesDOM	0.001

Tabulka 13: Srovnání průměrné doby v ms dotazování jednotlivých operací nad datovou sadou *LARGE*

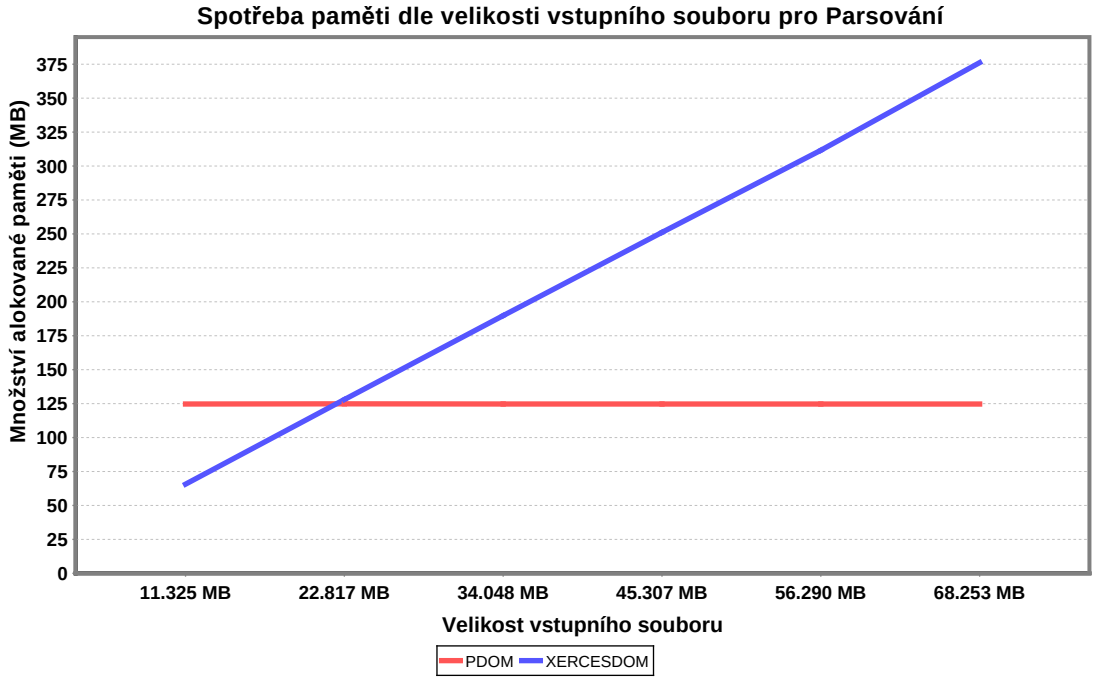


Obrázek 21: Srovnání průměrné doby dotazování (ve vteřinách) jednotlivých operací pro *XercesDOM* a *PDOM* nad datovou sadou *SMALL*



Obrázek 22: Srovnání průměrné doby dotazování (ve vteřinách) jednotlivých operací pro *XercesDOM* a *PDOM* nad datovou sadou *LARGE*

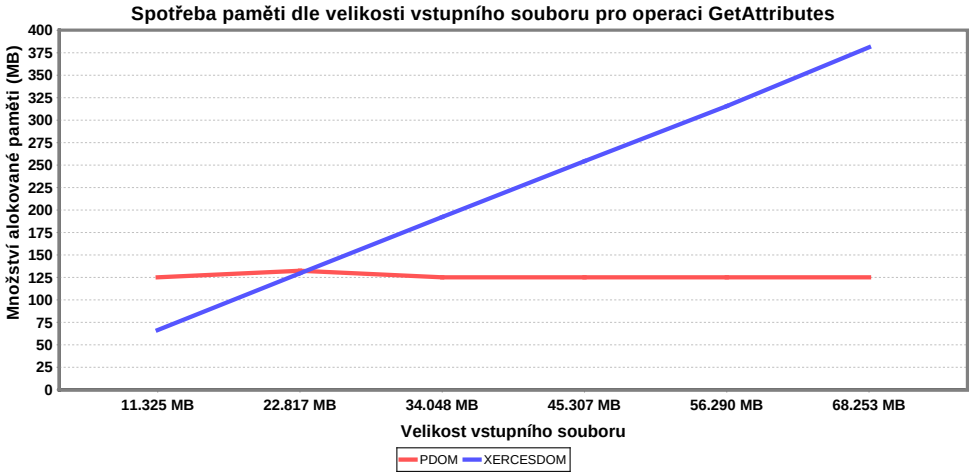
Další srovnávanou vlastností DOM je spotřeba paměti. V grafu 23 a tabulce 14 je vidět spotřeba paměti pro datovou sadu *SEQUENTION* pro parsování vstupního souboru. Z grafu je na první pohled jasné, že zatímco pro *PDOM* je spotřeba konstantních 125MB, které jsou využity v předalokovaných strukturách a cache přístupu na disk, vyžaduje *XercesDOM* poměrně rychle rostoucí množství paměti. Narozdíl od *XercesDOM*, který vyžaduje paměť pro uchování všech uzlů, je množství potřebné paměti pro *PDOM* závislé nikoli na celkové velikosti souboru, ale na jeho maximální hloubce. Výše uvedenému odpovídá i graf 24 a tabulka 15 vykonání operace *getAttributes*.



Obrázek 23: Spotřeba paměti pro parsování vstupního souboru v závislosti na velikosti vstupního souboru

Název souboru	Velikost vstupního souboru	Využitá paměť Xerces	Využitá paměť PDOM Dewey order	Využitá paměť PDOM Range order	Využitá paměť PDOM Containment order
output-0.1.xml	11.325 MB	65.672	124.715	124.715	124.676
output-0.2.xml	22.817 MB	128.000	124.844	124.848	124.801
output-0.3.xml	34.048 MB	189.824	124.723	124.719	124.680
output-0.4.xml	45.307 MB	251.129	124.707	124.703	124.668
output-0.5.xml	56.290 MB	311.586	124.703	124.703	124.660
output-0.6.xml	68.253 MB	376.164	124.703	124.707	124.668

Tabulka 14: Množství spotřebované paměti dle velikosti vstupního souboru pro operaci parsování



Obrázek 24: Spotřeba paměti pro parsování vstupního souboru v závislosti na velikosti vstupního souboru pro operaci GetAttributes

Název souboru	Velikost vstupního souboru	Využitá paměť paměť Xerces	Využitá paměť PDOM Dewey order	Využitá paměť PDOM Range order	Využitá paměť PDOM Containment order
output-0.1.xml	11.325 MB	66.430	125.129	125.160	124.547
output-0.2.xml	22.817 MB	129.602	132.422	125.145	124.535
output-0.3.xml	34.048 MB	192.270	125.117	124.543	124.543
output-0.4.xml	45.307 MB	254.387	125.121	125.148	124.539
output-0.5.xml	56.290 MB	315.648	125.125	125.156	124.547
output-0.6.xml	68.253 MB	381.145	125.117	125.152	125.129

Tabulka 15: Množství spotřebované paměti dle velikosti vstupního souboru pro operaci GetAttributes

Tato kapitola porovnala vlastnosti implementací *PDOM* a *XercesDOM*. Zatímco *XercesDOM* je efektivní při práci s malými soubory, kdy dosahuje výrazně rychlejšího rozparsování a mírně rychlejšího vyhodnocení dotazování než *PDOM*, není situace u větších souborů natolik jednoznačná. Z uvedených příkladů je jasné, že přestože dosahuje rychlost rozparsování XML souboru o přibližné velikosti 100MB jenom třetinu rychlosti rozparsování u *PDOM*, roste tento ukazatel u *XercesDOM* výrazně rychleji. Při porovnání dotazování mezi *XercesDOM* a *PDOM* nebylo možné dojít kvůli omezením implementace *XercesDOM*, která ovlivnila výsledky měření, k jednoznačnému závěru.

Srovnání měření nároku na paměť počítače přineslo očekávané výsledky. Zatímco množství paměti potřebné pro rozparsování vstupního souboru pro *PDOM* je v testovaných případech téměř konstantní, vykazuje *XercesDOM* lineární závislost, která odpovídá přibližně funkci $y = 6 * x$. Na základě analýzy zdrojového kódu aplikace je možné určit, že množství paměti potřebné k rozparsování vstupního souboru pro *PDOM* je přímo úměrné maximální hloubce zpracovávaného XML souboru. Při srovnání nároků na paměť počítače při vyhodnocení dotazování dosahovalo *PDOM* konstantní paměťové složitosti především díky využití struktur ukládajících zpracovávané elementy na disk. V případě *XercesDOM* je však paměťová náročnost lineárně závislá na velikosti vstupního souboru již kvůli nutnosti držet celý jeho rozparsovaný obsah v paměti počítače.

6 Závěr

V rámci této práce byla úspěšně implementována zjednodušená verze persistentního DOM, které poskytuje všechny důležité funkce pro navigaci ve stromu XML. Bylo dosaženo všech kritických cílů, které zahrnovaly implementaci jádra systému označeném jako *PersistentStorage*, jež umožňuje dva způsoby vyhodnocení dotazování, a tří číslovacích schémat. Toto obecně navržené jádro bylo následně využito k implementaci omezeného množství funkčnosti DOM, pojmenované jako tzv. elementární operace, a obohaceno o rozhraní v režimu příkazové řádky. V neposlední řadě bylo také představeno otestování výkonnosti jednotlivých konfigurací systému a srovnání s již existujícím DOM v podobě knihovny Xerces.

V průběhu práce bylo provedeno několik srovnávacích testování, při kterých bylo zjištěno mnoho zajímavých informací. Při srovnání konfigurace implementovaného řešení, se ukázalo nejefektivnější číslovací schéma *Containment order*, těsně následované schématem *Dewey order*. Nejméně efektivní se ukázalo číslovací schéma *Range order*, které se projevilo jako výrazně pomalejší u dotazů, jež ověřují vazbu mezi uzlem a jeho přímým rodičem. Při vyhodnocení dotazu se jako výkonnější ukázala preference persistentního pole namísto preference B-stromu. Nejznatelněji při srovnání množství přístupů k disku.

Srovnání doby vykonání dotazu mezi řešením postaveném na persistentním DOM a řešením využívajícím knihovny Xerces bohužel nepřineslo jednoznačně interpretovatelný výsledek. Implementace řešení využívající knihovny Xerces obsahovala omezení, které tomuto zhodnocení zabránila. Lepších výsledků bylo dosaženo při porovnání využití paměti. Ukázalo se, že paměťové nároky řešení, jež je postaveno na persistentním DOM, jsou téměř konstantní. Řešení, které využívá knihovny Xerces, však bylo lineárně rostoucí.

Přestože je uvedený systém v mnoha ohledech spíše hrubým náčrtem reálně nasaditelného řešení, bylo prakticky předvedeno, že se jedná o řešení principiálně kvalitní a funkční.

Realizace této práce přinesla autorovi mnoho nových poznatků ohledně návrhu klíčových systémů, využití návrhových vzorů a postupů pro optimalizaci řešení. Zejména kvůli nepříliš kvalitnímu úvodnímu návrhu si autor prakticky ověřil jeho důležitost a měl tak možnost získat cenné zkušenosti s návrhem podobných softwarových řešení. Přestože byl úvodní návrh řešení částečně přepracován, zůstalo v kódu aplikace několik míst, která jsou zbytečně složitá či neefektivní. Zmíněné částečné přepracování je však odstranilo z klíčových částí systému, což umožnilo zdárné dokončení této práce.

Představená implementace persistentního DOM by mohla těžit z řady případných rozšíření. Prvním zajímavým rozšířením by byla implementace již zmíněného rozhraní v podobě standardu XPATH a implementace podpory komprimace použitých datových struktur. Vhodným rozšířením by také byla implementace alternativního parseru vstupního XML namísto použitého parseru z knihovny *Xerces*, což by přineslo možnost srovnání parsovacího mechanismu a zhodnocení jeho případné další optimalizace. Dalším velkým přínosem by bylo využití aktuálního testování k další optimalizaci dotazování. Zatímco implementace vlastního persistentního DOM se jevila jako velmi efektivní, srovnávané řešení v podobě implementace založené na knihovně Xerces nedosahovala vždy

srovnatelné výkonnosti. Zajímavým pokusem by byla adopce jiného řešení k srovnání založené na jiné knihovně klasického DOM, alternativně přímo srovnání s nějakou XML databází či implementací persistentního DOM.

7 Reference

[1] KIM, Seung Min, Suk I. YOO, Eunji HONG, Tae Gwon KIM a Il Kon KIM. A document object modeling method to retrieve data from a very large XML document. *Proceedings of the 2007 ACM symposium on Document engineering - DocEng '07*. New York, New York, USA: ACM Press, 2007, s. 59-68. DOI: 10.1145/1284420.1284439. Dostupné z: http://dl.acm.org/citation.cfm?doid=1284420.1284439

[2] KIM, Seung Min a Suk I. YOO. DOM tree browsing of a very large XML document: Design and implementation. *Journal of Systems and Software*. 2009, vol. 82, issue 11, s. 1843-1858. DOI: 10.1016/j.jss.2009.05.043. Dostupné z: http://linkinghub.elsevier.com/retrieve/pii/S0164121209001216

[3] DELPRATT, O'Neil, Rajeev RAMAN a Naila RAHMAN. Engineering succinct DOM. *Proceedings of the 11th international conference on Extending database technology Advances in database technology - EDBT '08*. New York, New York, USA: ACM Press, 2008, s. 49-. DOI: 10.1145/1353343.1353354. Dostupné z: http://portal.acm.org/citation.cfm?doid=1353343.1353354

[4] ZHANG, Ning, Varun KACHOLIA a M. Tamer ÖZSU. A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML. *Proceedings of the 20th International Conference on Data Engineering*. 2004, ICDE '04, s. 54-. Dostupné z: http://dl.acm.org/citation.cfm?id=977401.978103

[5] WONG, Raymond K., Franky LAM a William M. SHUI. Querying and maintaining a compact XML storage. *Proceedings of the 16th international conference on World Wide Web - WWW '07*. New York, New York, USA: ACM Press, 2007, WWW '07, s. 1073-. DOI: 10.1145/1242572.1242717. Dostupné z: http://portal.acm.org/citation.cfm?doid=1242572.1242717

[6] LI, Quanzhong a Bongki MOON. Indexing and Querying XML Data for Regular Path Expressions. *Proceedings of the 27th International Conference on Very Large Data Bases*. 2001, VLDB '01, s. 361-370. Dostupné z: http://dl.acm.org/citation.cfm?id=645927.672035

[7] TATARINOV, Igor, Stratis D. VIGLAS, Kevin BEYER, Jayavel SHANMUGASUNDARAM, Eugene SHEKITA a Chun ZHANG. Storing and querying ordered XML using a relational database system. *Proceedings of the 2002 ACM SIGMOD international conference on Management of data - SIGMOD '02*. New York, New York, USA: ACM Press, 2002, s. 204-. DOI: 10.1145/564691.564715. Dostupné z: http://portal.acm.org/citation.cfm?doid=564691.564715

[8] ZHANG, Chun, Jeffrey NAUGHTON, David DEWITT, Qiong LUO a Guy LOHMAN. On supporting containment queries in relational database management systems. *Proceedings of the 2001 ACM SIGMOD international conference on Management of data - SIGMOD '01*. New York, New York,

USA: ACM Press, 2001, s. 425-436. DOI: 10.1145/375663.375722. Dostupné z: http://portal.acm.org/citation.cfm?doid=375663.375722

[9] BAČA, Radim, Jiří WALDER, Martin PAWLAS a Michal KRÁTKÝ. Benchmarking the compression of XML node streams. *Proceedings of the 15th international conference on Database systems for advanced applications*. 2010, DASFAA'10. Dostupné z: http://dl.acm.org/citation.cfm?id=1880853.1880876

[10] ANDERSON, Tim. Introducing XML. *Tim Anderson's ITWriting* [online]. 2004 [cit. 2013-07-21]. Dostupné z: http://www.itwriting.com/xmlintro.php

[11] Document Object Model (DOM). W3C. *W3C* [online]. 2005 [cit. 2013-07-21]. Dostupné z: http://www.w3.org/DOM/

[12] Xerces. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2013-07-21]. Dostupné z: http://en.wikipedia.org/wiki/Xerces

[13] MSXML. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2013-07-21]. Dostupné z: http://en.wikipedia.org/wiki/MSXML

[14] Probst, Martin. *Processing Arbitrarily Large XML using a Persistent DOM.* Presented at Balisage: The Markup Conference 2010, Montréal, Canada, August 3 - 6, 2010. In Proceedings of Balisage: The Markup Conference 2010. Balisage Series on Markup Technologies, vol. 5 (2010). doi:10.4242/BalisageVol5.Probst01.

[15] *ANNOUNCE: Update of GMD-IPSI XQL engine* [Original annoucment email]. 7.5.1999. 1999 [cit. 22.7.2013]. Dostupné z: http://lists.w3.org/Archives/Public/www-dom/1999AprJun/0056.html

[16] *Ozone prostřednictvím WayBackMachine* [online]. 2011 [cit. 2013-07-22]. Dostupné z: http://web.archive.org/web/20110902221037/http://ozone-db.org/frames/home/what.html

[17] MEIER, Wolfgang. EXist: An Open Source Native XML Database. *Revised Papers from the NODe 2002 Web and Database-Related Workshops on Web, Web-Services, and Database Systems*. Editor Alicja Szymczakowa, Jan Szymczak. 2003. Dostupné z: http://dl.acm.org/citation.cfm?id=648032.744076

[18] XML. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2013-07-21]. Dostupné z: http://en.wikipedia.org/wiki/XML

[19] ARCINIEGAS, Fabio A. XML Programming with C++. *XML.com* [online]. 1999 [cit. 2013-07-21]. Dostupné z: http://www.xml.com/lpt/a/475

[20] *Apache Xerces* [online]. 2013 [cit. 2013-07-21]. Dostupné z: http://xerces.apache.org/

[21] *The XML C parser and toolkit of Gnome: libxml* [online]. [cit. 2013-07-21]. Dostupné z: http://xmlsoft.org/

[22] MSXML. MICROSOFT. *MSDN* [online]. 2013 [cit. 2013-07-21]. Dostupné z: http://msdn.microsoft.com/en-us/library/ms763742.aspx

[23] NEUBAUER, Peter. B-Trees: Balanced Tree Data Structures. *Bluer White* [online]. 1999 [cit. 2013-07-21]. Dostupné z: http://www.bluerwhite.org/btree/

[24] Big data. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2013-07-22]. Dostupné z: http://en.wikipedia.org/wiki/Big_data

[25] *Database Reasearch Group: Data structures, Query processing, XML, ...* [online]. © 2010 - [cit. 2013-07-30]. Dostupné z: http://db.cs.vsb.cz

Seznam příloh

Příloha A: Kompilace aplikace

Aplikace byla vyvíjena v prostředí aplikace *Visual Studio 2010*.

Pro kompilaci je nejdříve třeba stáhnout framework z adresy `https://dbedu.cs.vsb.cz:10001/amphora/2.3` v revizi *4113*.

Po stažení tohoto frameworku je třeba zdrojové soubory předané s touto prací umístit do adresáře `src\test\mbt\roz0015`, přičemž případné soubory v tomto adresáři je třeba přepsat, jelikož mohou obsahovat starší verzi aplikace.

Následně je třeba instalovat do aplikace knihovnu Xerces-C++ ve 32bitové verzi 3.1.1 pro *Visual Studio 2010*, jež lze získat například na adrese `http://apache.miloslavbrada.cz/xerces/c/3/binaries/xerces-c-3.1.1-x86-windows-vc-10.0.zip`.

Instalaci proveďte pomocí následujícího postupu:

- Stáhněte instalační balík ze zmíněného umístění.
- Nakopírujte obsah adresáře `include\xercesc` ze zmíněného balíku do `src\test\mbt\roz0015\xercesc`.
- Nakopírujte celý adresář `lib` ze zmíněného balíku do `src\test\mbt\roz0015\xercesc`.
- Nakopírujte soubory `bin\xerces-c_3_1.dll` a `bin\xerces-c_3_1D.dll` do adresářů `src\test\mbt\roz0015\test\PDOM\Debug` a `src\test\mbt\roz0015\test\PDOM\Release`. V případě, že tyto adresáře neexistují, je třeba je vytvořit.

Vlastní soubor projektu lze nalézt v umístění `src\test\mbt\roz0015\test\PDOM\PDOM.sln`. Obsahuje všechna potřebná nastavení k realizaci kompilace jak v režimu *Debug*, tak v režimu *Release*.

Příloha B: Použití aplikace

Aplikace vyžaduje zadání následujících argumentů:

```
Usage: PDOM.exe pdom ordering parse input_file dataFile [depth]
Usage: PDOM.exe pdom ordering preference query inputFile dataFile
[startNodeOrdering] querystring
Usage: PDOM.exe pdomX ordering preference query inputFile dataFile
[startNodeOrdering] querystring
Usage: PDOM.exe xerces filename [startNode] querystring
Usage: PDOM.exe xercesX filename [startNode] querystring
        domtype: xerces pdom xercesX pdomX
        ordering: 1 - Containment order, 2 - Range order,
  3 - Dewey order
        preference: 0 - Btree query preference, 1 - Persistent
  array query preference
        querystring: operation[node_name,node_type,value]
        node_type: attribute, node, value
        operation: getDescendants, getDescendantsByTagName,
   getChildren, getChildrenByTagName, getAttributes,
   getValue, getSiblings
        xercesX and pdomX do save results of measurement.
```

Jak lze vidět z výpisu použití aplikace, je možné ji spustit ve dvou režimech - *pdom* (respektive *pdomX*) a *xerces* (respektive *xercesX*). V režimu *pdom* aplikace realizuje dotazy s využitím persistentního DOM navrženého a implementovaného v rámci této práce, zatímco v režimu *xerces* jsou využívány DOM metody knihovny Xerces. Rozhraní pro volání metod je záměrně realizováno podobným způsobem. Varianty režimů se znakem *X* na konci reprezentují volání, při kterém nejsou ukládány výsledky o měření (ty jsou ukládány vždy v pracovním adresáři). Tyto speciální varianty byly přidány pro podporu srovnávacího testování aplikace.

V režimu *pdom* realizuje aplikace buď rozparsování dokumentu do úložiště (při zadání argumentu *parse*) či dotazování vůči dokumentu v úložišti (při zadání argumentu *query*).

Při parsování dokumentu je nutné zadat vstupní argumenty *ordering*, *inputFile*, *dataFile* a je možné zadat i přepínač *depth*.

Za argument *ordering* je třeba zadat následující možné hodnoty:

- **1** - pro číslovací schéma *Containment order*.
- **2** - pro číslovací schéma *Range order*.
- **3** - pro číslovací schéma *Dewey order*.

Místo argumentů *inputFile* a *dataFile* je třeba zadat cestu k existujícímu vstupnímu XML souboru a cílovému datovému souboru. Pozor - datový soubor má vždy automaticky připojenu příponu „.dat“.

Pokud je specifikováno argumentem *ordering* číslovací schéma *Dewey order*, je možné zadat v argumentu *depth* maximální možnou hloubku vstupního XML souboru. To umožňuje vytvoření úložiště, do kterého lze zpracovat soubory s vetší hloubkou, než je první zpracovaný soubor. Pokud není tento argument zadán, je hloubka zjištěna ze vstupního souboru automaticky. Pozor - použitá hloubka ovlivňuje velikost datového souboru.

Při parsování vstupního XML dokumentu může v některých případech dojít k poškození datového souboru. Vzhledem k podstatě číslovacích schémat není vždy možné korektně ověřit, že je dokument vytvářen se stejným číslovacím schématem, jaké je použito v datovém souboru. V takovémto případě parsování uspěje, ale selžou všechny pokusy o dotazování vůči úložišti bez ohledu na nastavení číslovacího schématu. Proto je při práci s konkrétním datovým souborem důležité používat vždy pouze jedno číslovací schéma.

Při dotazování vůči úložišti je třeba zadat vstupní argumenty *ordering*, *preference*, *inputFile*, *dataFile*, *querystring* a je možné zadat i argument *startNodeOrdering*.

Argumenty *ordering*, *inputFile* a *dataFile* jsou stejné jako při procesu parsování XML dokumentu. Liší se pouze použitím v aplikaci. Soubor předaný v argumentu *inputFile* již nemusí existovat, jelikož je použit pouze k identifikaci souboru v datovém úložišti.

Argument *preference* specifikuje způsob vyhodnocení dotazu a může nabývat následujících hodnot:

- **0** - pro vyhodnocení dotazu s *preferencí použití B-stromu*.
- **1** - pro vyhodnocení dotazu s *preferencí použití persistentního pole*.

Volitelný agument *startNodeOrdering* umožňuje specifikovat uzel, na který je dotaz kladen pomocí identifikátoru generovaným použitým číslovacím schématem. Tento atribut se zadává ve formátu `(číslo,číslo,číslo,...)` dle konkrétní ho číslovacího schématu. Například při použití číslovacího schématu *Dewey order* může vypadat takto: `(1,1,10,5)`. Není-li tento argument zadán, realizují se všechny dotazy vůči kořenovému uzlu dokumentu.

Argument *queryString* specifikuje typ dotazu a jeho argumenty, které jsou specifikovány ve formátu `typ_dotazu(argumentA=hodnotaA,argumentB=hodnotaB,...)`.

Argumenty jsou definovány následovně:

1. **tagName** - určuje očekávaný název uzlu. Lze použít pouze u operací *getDescendantsByTagName* a *getChildrenByTagName*.
2. **tagType** - lze použít pro zadání očekávaného typu uzlu. Lze použít hodnoty *node*, *attribute*, *value*, *any*.
3. **value** - specifikuje očekávanou hodnotu uzlu.

Typ dotazu může nabývat následujících hodnot:

1. **getDescendants** - provede pokus o načtení všech potomků uzlu. Bez zadání dalších argumentů zpracovává pouze elementy typu *node*.
2. **getDescendantsByTagName** - provede pokus o načtení všech potomků uzlu s specifickým jménem uzlu. Vyžaduje vždy specifikaci atributu *tagName*. Bez zadání dalších argumentů zpracovává pouze elementy typu *node*.
3. **getChildren** - provede pokus o načtení všech přímých potomků uzlu. Bez zadání dalších argumentů zpracovává pouze elementy typu *node*.
4. **getChildrenByTagName** - provede pokus o načtení všech přímých potomků uzlu se specifickým jménem uzlu. Vyžaduje vždy specifikaci atributu *tagName*. Bez zadání dalších argumentů zpracovává pouze elementy typu *node*.
5. **getAttributes** - provede pokus o načtení všech atributů uzlu.
6. **getValue** - vrátí hodnotu aktuálního uzlu. Tomuto atributu jako jedinému nelze specifikovat žádné argumenty.
7. **getSiblings** - načte sourozence zadaného uzlu.

Při zpracování souboru lze zadat vždy pouze jednu operaci k provedení.

Režim zpracování *xerces* podporuje ze své podstaty pouze dotazování, jež je prováděno vůči fyzickému souboru. Ten je při každém volání znovu rozparsován do paměti.

V tomto režimu jsou přijímány argumenty *filename*, *querystring* a volitelný argument *startNode*. Argument *filename* slouží k zadní vstupního souboru, vůči kterému jsou dotazy vykonány. Argument *querystring* přjímá identickou syntax jako v případě režimu *pdom*.

Volitelný argument *startNode* slouží ke stejnému účelu jako argument *startNodeOrdering* v režimu *pdom*. Uzel vůči, kterému je následně vykonána operace, však není určen číslovacím schématem, ale identifikátorem uzlu po vykonání operace *getDescendants* či operace *getChildren* s výchozími parametry. Identifikátor je následně zadán ve formátu `[identifikace_operace][index_uzlu]`, kde *identifikace_operace* nabývá hodnot *d* či *c* dle použité operace a *index_uzlu* obsahuje pořadí určeného uzlu v navrácených výsledcích. Příkladem takového identifikátoru může být například `d365`.

Příloha C: Obsah přiloženého CD.

Přiložené CD obsahuje následující adresáře.

1. **build** - obsahuje poslední sestavení aplikace.
2. **doc** - obsahuje text vlastní práce.
3. **measurements** - obsahuje soubory s měřeními, které byly použity pro generování grafů a tabulek v práci.
4. **sources** - zdrojové soubory aplikace. Pro správnou funkčnost vyžadují framework z URL `https://dbedu.cs.vsb.cz:10001/amphora/2.3` a aktuální verzi *Apache Xerces*.
5. **testing_data** - obsahuje testovací data, která jsou zmíněna v práci.